Grady Booch



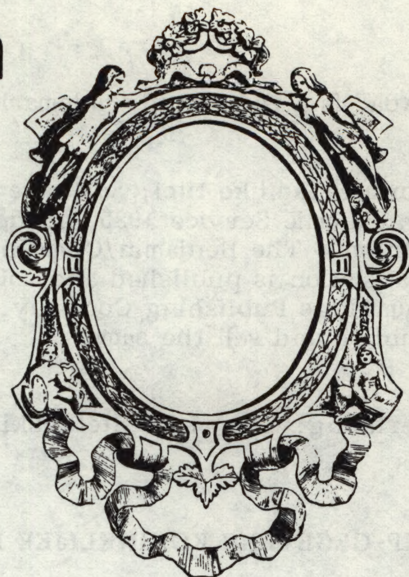
Systeemontwikkeling met Ada®

ACADEMIC SERVICE

SYSTEEMONTWIKKELING MET ADA

Voor Jan

Grady Booch



Systeemontwikkeling met Ada®

ACADEMIC SERVICE

Oorspronkelijke titel: *Software engineering with Ada.*

Vertaling: drs. M.M. Stefanski

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Booch, Grady

Systeemontwikkeling met Ada / Grady Booch : [vert. uit het Engels door M.M. Stefanski]. - Den Haag : Academic Service. - I11.
Vert. van: Software engineering with Ada. - Menlo Park, Calif. : Benjamin/Cummings, 1983. - (Benjamin/Cummings series in computing and information sciences). - Met lit. opg., reg.
ISBN 90-6233-133-5
SISO 365.3 SVS 8.12.3 UDC 681.3.06+800.92 ADA UGI 650
Trefw.: ADA (programmeertaal) / programmeren (comp.).

Uitgegeven door: Academic Service
Postbus 96996
2509 JJ Den Haag
Zetwerk: INFOTYPE Maarheeze
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
Omslagontwerp: JAM Gauw
ISBN 90-6233-133-5

INHOUD

VOORWOORD BIJ DE NEDERLANDSE VERTALING

In het computerjargon zijn nogal wat Engelse woorden ingeburgerd. Woorden als computer, software engineering, interface, data base, editor, default en display worden meestal niet vertaald. Nu is het voor een vertaler wel een uitdaging Nederlandse equivalenten te vinden voor deze termen (bijvoorbeeld 'koppelvlak' voor 'interface' of 'verstekwaarde' voor 'default value'), maar de zin hiervan is twijfelachtig, omdat de kans op algemene verbreiding van dergelijke taalpurismen klein is.

In deze vertaling is de volgende strategie toegepast: begrippen als 'embedded system' en 'interface' worden in het begin van het boek enige malen omschreven, maar later wordt in arren moede toch maar weer 'interface' gebruikt, om het (de?) interface van de lezer met zijn vakgenoten niet al te zeer te verstoren. Bij de vertaling is gebruik gemaakt van de door het Nederlands Normalisatie Instituut uitgegeven *Ontwerp Woordenlijst Informatica* (1983). Ook deze, toch zeer uitgebreide lijst kon ons echter voor de vertaling van woorden als 'editor', 'base line', 'default', 'embedded system' en 'prompt' geen uitkomst bieden.

De vertaling van de titel van dit boek: *Software Engineering with Ada* vat het probleem van de vertaler nog eens samen. Juist omdat de term 'software engineering' meestal niet vertaald wordt, lopen de meningen over wat er precies onder verstaan moet worden uiteen. 'Software' is, volgens de woordenlijst van het NNI, 'programmatuur' in de zin van een 'intellectuele schepping bestaande uit programma's, procedures, regels en bijbehorende documentatie voor een gegevensverwerkend systeem'. 'Engineering' is wat de ingenieur doet en dat is het realiseren van kunstig bedachte en meestal ingewikkelde constructies en mechanismen.

'Software engineering' zou dus het best met 'programmeren' vertaald kunnen worden, want dat is al moeilijk genoeg. 'Programmeren' wordt echter door velen om de een of andere reden beschouwd als een te zwakke omschrijving voor het 'kunstig bedenken van complexe mechanismen voor het verwerken van gegevens' en 'Programmeren met Ada' zou daarom onvoldoende de inhoud van dit boek dekken. Om toch het werk van de vertaler niet te beperken tot de vertaling van het woordje 'with', werd gekozen voor *Systeemontwikkeling met Ada*, een titel die iets meer wijst in de richting van het ontwerpen en bouwen van grote programma's en de problemen die daarbij te pas komen.

Gouda, augustus 1984 — Maarten Stefanski

VOORWOORD

Leren programmeren in Ada is een uitdaging, want Ada is niet zomaar een programmeertaal. In Ada zijn alle moderne ideeën over programmeren én over de bouw van grote systemen samengebracht.

Ada is niet alleen een programmeertaal, maar een programmeersysteem dat een belangrijke invloed zal hebben op de methoden voor software-ontwikkeling in de komende jaren.

De invloed van computers op onze maatschappij is wel vergeleken met die van de Industriële Revolutie. Zowel bij de overheid als in de industrie vindt de automatisering steeds meer toepassingen en de gevolgen daarvan worden voor iedereen merkbaar. In de komende jaren zal onze samenleving door de automatisering nog verder veranderen. Nu al bevatten auto's, wasmachines, horloges en spelletjes microprocessoren dankzij de chiptechnologie. Niet alleen hobbyïsten, maar ook leken op computergebied schaffen zich 'personal computers' aan voor hun financiële administratie, regeling van de centrale verwarming en zelfs voor het invullen van hun belastingformulier.

Elke computer moet worden geprogrammeerd en hoe meer computers er komen des te meer neemt de, nu al zeer grote, vraag naar programmatuur toe. In de komende jaren zal software-ontwikkeling een der belangrijkste technologieën zijn in onze samenleving. En het gaat hier niet om zo maar een programma schrijven, nee het gaat om het bouwen van complexe softwaresystemen.

Eind jaren zestig en gedurende de zeventiger jaren is een begin gemaakt met wat genoemd wordt de techniek van 'software engineering', dat wil zeggen de techniek van het bouwen van programmasystemen die een groot aantal, veelal ingewikkelde functies uitvoeren. Men begon toen het evangelie van ontwerpmethodieken, gestructureerd programmeren en het beheersen van de softwarelevenscyclus uit te dragen. Er werd onderzoek gedaan naar nieuwe technieken, speciale talen en gereedschappen voor software-ontwikkeling. Doel was de kwaliteit van de programmatuur te vergroten en de produktiviteit te verhogen van degenen die de programma's moesten ontwikkelen.

Al in 1975 zagen enkele vooruitziende medewerkers van het departement van defensie in de Verenigde Staten in, dat er op korte termijn behoefte zou ontstaan aan een krachtig software-ontwikkelingssysteem, gebaseerd op een standaardprogrammeertaal die alle moderne ideeën over programmeren zou omvatten. Toen een programma

van eisen was geformuleerd bleek dat geen enkele bestaande programmeertaal aan die eisen voldeed en daarom werd een wedstrijd in het ontwerpen van zo'n taal uitgeschreven. De winnaar werd Ada.

De hele ontwikkeling van Ada werd ondersteund door de ideeën en adviezen van een grote groep vooraanstaande computerwetenschapsmensen uit de hele wereld. Enthousiast door de mogelijkheid mee te werken aan het maken van een taal die zou kunnen leiden tot het invoeren van verfijndere software-ontwikkelingsmethoden werkten velen zonder enige vergoeding mee en wisselden ervaringen uit. Zelfs als er kritiek geleverd werd was die in de meeste gevallen opbouwend.

Het resultaat is een taal die in de wereld van de computerwetenschap grote belangstelling geniet, en die wel de programmeertaal van de jaren tachtig wordt genoemd. Volmaakt is de taal zeker niet en het laatste woord in de ontwikkeling van programmeertalen is ook nog niet gesproken. In sommige gevallen is gezocht naar compromisoplossingen, maar de grondslagen van verantwoord systeemontwerp werden geen geweld aangedaan. Het ziet er naar uit dat met behulp van Ada, ontworpen als de taal is voor het overdraagbaar maken van programmatuur, een industrie voor de produktie van softwarecomponenten van de grond kan komen. De mogelijkheid om met behulp van het Ada package-begrip gegevens beperkt toegankelijk te maken en om abstracte datastructuren te creëren, kan de ontwikkeling van programmabibliotheken en andere vormen van algemeen toepasbare software zeer zeker bevorderen.

Dit boek behandelt een groot deel van de mogelijkheden van Ada voor de systeembouw. Het boek gaat uit van een vaste ontwerpmethode en probeert de ontwikkeling van een goede programmeerstijl te bevorderen. De schrijver is een ervaren docent in de taal Ada. Hij heeft een groot aantal cursussen gegeven voor het Amerikaanse departement van defensie en de ervaring daar opgedaan heeft hem geleerd hoe de leerstof het best kan worden ingedeeld. Dit heeft geleid tot een evenwichtige inleiding in de taal en, belangrijker, in correct gebruiken ervan.

Als u Ada begint te bestuderen als zo maar een nieuwe programmeertaal, dan zult u teleurgesteld zijn. U zult de taal dan ingewikkeld vinden en uw programma's zullen slecht geconstrueerd zijn. Als u daarentegen het leren van Ada beschouwt als een hulpmiddel voor een hoger doel: inzicht verkrijgen in de methoden en technieken van software-ontwikkeling, dan zal blijken dat de taal door zijn structuur eenvoudiger is dan door zijn omvang op het eerste gezicht wordt gesuggereerd.

Formuleren van correcte en overzichtelijke ontwerpen is het uiteindelijke doel. Dit boek wil u bij het streven daarnaar helpen en zal aangeven hoe Ada U daarbij kan ondersteunen.

Larry E. Druffel
Arlington, Virginia
januari, 1983

INLEIDING

Ada is een programmeertaal voor algemeen gebruik, waarin ontwikkelde oplossingen bondig kunnen worden uitgedrukt. De taal werd gemaakt naar aanleiding van een initiatief van het departement van defensie van de Verenigde Staten als een antwoord op de 'Software-crisis'. Ada werd vooral ontworpen voor grote procesbesturingssystemen, maar ook in andere gebieden zal Ada zeker toepasbaar blijken.

In tegenstelling tot andere hogere programmeertalen zoals FORTRAN, COBOL en zelfs Pascal, bevat Ada een vrij complete gereedschapset voor het ontwikkelen van softwaresystemen en de programmeur wordt er bijna toe gedwongen dit gereedschap ook te gebruiken. Zinvol toepassen van deze gemeenschappelijke poging tot het creëren van een nieuwe programmeertaal betekent daarom het combineren van goede ontwerptechnieken met de uitdrukkingsmogelijkheden van Ada. Met behulp van Ada kunnen de helderheid, betrouwbaarheid, de doelmatigheid en de onderhoudbaarheid van softwaresystemen aanmerkelijk worden verbeterd.

Ada is niet alleen een programmeertaal; met behulp van Ada's programmeeromgeving (APSE: Ada Programming Support Environment), kunnen problemen beter worden geanalyseerd en kunnen oplossingsmethoden worden geformuleerd die direct aansluiten bij de problemen, zoals die in de realiteit worden waargenomen.

Het Doel Van Dit Boek

Dit boek wil meer zijn dan een inleiding in de programmeertaal Ada. De schrijver heeft de volgende doelstellingen voor ogen gehad:

- geven van een gedetailleerde beschrijving van Ada;
- door middel van voorbeelden duidelijk maken wat een goed ontwerp en wat een goede programmeerstijl in Ada betekent;
- een object-gerichte ontwerpmethode presenteren die van de mogelijkheden van Ada gebruik maakt en waarmee de complexiteit van grote softwaresystemen beheersbaar blijft.

Kortom, de bedoeling is te laten zien hoe Ada gebruikt kan worden ter ondersteuning van het ontwerp en de produktie van softwaresystemen.

Toen Ada ontwikkeld werd volgden op de voorlopige taalbeschrijving twee versies van de volledige beschrijving, en wel in juli en november 1980. In 1981 werden nog enkele kleine wijzigingen aangebracht op grond van de beoordeling van het Americal National Standards Institute (ANSI). De tekst in dit boek is aan deze laatste versie aangepast en voldoet dus aan de ANSI-standaard voor Ada. De meeste programmavoorbeelden zijn verder daadwerkelijk gecompileerd en gedraaid om onjuistheden te elimineren.

Het boek is een neerslag van ervaring, opgedaan bij de United States Air Force Academy en het AJPO (Ada Joint Program Office) tijdens onderwijs in Ada aan meer dan 2500 studenten. Ook werden presentaties gegeven aan groepen met uiteenlopende achtergrond: leken op programmeergebied, beginnende studenten, gevorderden, beroepsprogrammeurs en projectleiders. Hierdoor zijn verschillende presentatiemethodes getest en vooral met de behoeften van beroepsmatige software-ontwerpers is rekening gehouden.

Het boek kan dienen als een volledige beschrijving van Ada, zowel voor programmeurs die systemen in Ada willen ontwerpen, als voor managers en projectleiders die inzicht willen krijgen in het gebruik van dit krachtige gereedschap. Omdat de belangstelling van deze twee groepen niet helemaal parallel loopt zijn onderdelen die speciaal voor managers interessant zijn met een Ⓜ gemerkt, en onderdelen van specifiek belang voor programmeurs met een Ⓟ. Kennis van de elementaire beginselen van het programmeren wordt verondersteld.

Over De Inhoud

Structuur

Veel beschrijvingen van programmeertalen beperken zich tot de syntactische (hoe luiden de uitdrukkingsregels?) en semantische (wat betekent een uitdrukking?) aspecten van de taal. In dit boek is gekozen voor een benadering vanuit het ontwerp van programmatuur. Vervolgens wordt Ada volgens de methode van afnemende abstractie (een top-down methode) geïntroduceerd, analoog aan de bestaande methodes voor het ontwerpen van goede programma's. De methode komt overeen met de aanbevelingen van de ACM SIGPLAN groep (Association for Computing Machinery, Special Interest Group for Programming Languages), gedaan tijdens het in december 1980 in Boston gehouden Ada symposium. De methode sluit ook aan bij de AJPO (Ada Joint Program Office) filosofie over onderwijs in Ada.

Het boek werd onderverdeeld in acht 'pakketten' (overeenkomstig het Ada packagebegrip), die ieder weer zijn opgebouwd uit drie logisch samenhangende hoofdstukken. Het eerste pakket gaat in op het probleemgebied waarvoor Ada oplossingen kan bieden. De ontwikkelingsgeschiedenis van Ada wordt nader bekeken om enige achtergrondideeën te geven over de betekenis van Ada's specifieke eigenschappen.

Het tweede pakket bevat een beschrijving van enige grondslagen voor het ontwerpen van software en van de recente ontwikkelingen op dat gebied. Daarna wordt een inleiding gegeven in de object georiënteerde ontwerpmethode.

In pakket drie tot en met zeven wordt Ada uitgebreid geïntroduceerd als de praktische oplossing voor de eerder beschreven methodes. Dit wordt toegelicht met een vijftal voorbeelden met toenemende moeilijkheidsgraad. Als alle voorbeelden worden doorgewerkt dan heeft men met vrijwel alle karakteristieke kenmerken van Ada kennis gemaakt. De problemen zijn bovendien bedoeld als illustratie van de object-georiënteerde ontwerpmethode is er is naar gestreefd de oplossingen te formuleren in een heldere en begrijpelijke programmeerstijl. De hoofdstukken tussen de vijf grote voorbeeldproblemen bevatten diepgaande beschrijvingen van Ada's taalconstructies.

Tenslotte wordt in pakket acht de Ada programmeeromgeving (APSE) beschreven en de toepassing van Ada gedurende de hele softwarelevenscyclus.

Ondersteuning

De meeste hoofdstukken worden afgesloten met een aantal opgaven. Verder zijn er zeven aanhangsels toegevoegd met technische gegevens over Ada: twee met syntaxdiagrammen en beknopte aanwijzingen over programmeerstijl; drie over de aan de taal toegevoegde standaardfuncties en twee met oplossingen voor de problemen en oefeningen.

Moeilijke opgaven zijn met een * gemerkt. Het boek wordt afgesloten met een verklarende Ada woordenlijst en een uitgebreide bibliografie.

Organisatie

Het boek kan op een aantal niveaus gelezen worden. Bij gebruik voor een cursus van totaal 40 college-uren zou de volgende indeling kunnen worden gevolgd:

Uur	Hoofdstuk	Onderwerp
1	1	Inleiding
2	2	De softwarecrisis
3-4	3	Ada's ontwikkelingsgeschiedenis

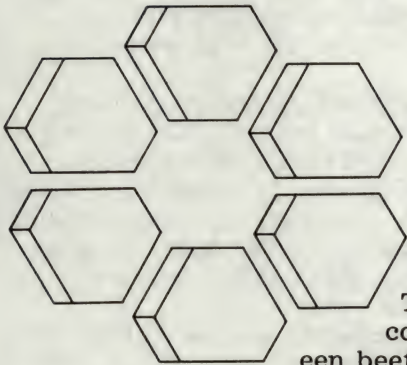
Uur	Hoofdstuk	Onderwerp
5-8	4	Software ontwerpmethoden
9-10	5	Object-georiënteerd ontwerp
11-12	6	Ada, een overzicht
13	7	Eerste probleem
14-15	8	Abstract voorstellen van gegevens en types in Ada
16	9	Tweede probleem
17-18	10	Subprogramma's
19-20	11	Expressies en instructies
21	12	Tweede probleem (vervolg)
22-23	13	Packages
24-25	14	Generieke programma-eenheden
26	15	Derde probleem
27-28	16	Taken
29-30	17	Opvangen van uitzonderingen en speciale elementaire eigenschappen
31	18	Vierde probleem
32	19	Input/output
33-34	20	Programmeren in het groot
35	21	Vijfde probleem
36-37	22	De Ada programmeeromgeving APSE
38-39	23	Ada en de softwarelevenscyclus
40	24	Verdere ontwikkelingen en conclusies

Bij wijze van korte introductie voor projectmanagers wordt de volgende indeling voorgesteld:

Blok 1: hoofdstuk 2 en 3
Blok 2: hoofdstuk 4 en 23
Blok 3: hoofdstuk 6
Blok 4: hoofdstuk 21, 22 en 24

Pakket 1

PROBLEEMSTELLING



Zolang er geen computers bestonden was programmeren geen probleem. Toen we een paar niet erg krachtige computers hadden werd programmeren een beetje problematisch. Nu we gigantische computers hebben is programmeren ook een gigantisch probleem geworden. In deze zin heeft de elektronica-industrie geen enkel probleem opgelost, maar alleen problemen geschapen: hoe datgene te gebruiken wat deze industrie geproduceerd heeft.

E.W. Dijkstra
Turing Award Lecture, 1972 [1]

1 INLEIDING

De mens is van nature een gereedschapsmaker en een gereedschapsgebruiker. Als wij de ontwikkelingsgeschiedenis van de mensheid bestuderen dan blijkt vaak dat het gebruik van een nieuw of krachtiger technisch of sociaal gereedschap omwentelingen heeft veroorzaakt. Zo heeft de introductie van het geschreven woord en later van het woord in (televisie)beeld onze samenleving zonder twijfel belangrijk veranderd. Instrumenten van de medische wetenschap, zoals de microscoop en röntgenstralen hebben levens gered, net zoals het gereedschap van de kunstenaar, muziekinstrumenten, penseel, levens verrijkt heeft. Steeds werden gereedschappen gemaakt of verfijnd om in een bepaalde behoefte te voorzien. Met behulp van gereedschap kunnen werkzaamheden vaak efficiënter worden verricht en vaak zou het werk onmogelijk zijn zonder een bepaald stuk gereedschap.

Vergeleken bij andere vakken is informatica een jonge wetenschap. Toch heeft het vak al zijn revoluties gekend, die ruwweg parallel liepen met de zogenaamde generaties van computerapparatuur, ontwikkeld dankzij het gereedschap van de computerarchitect. Gereedschap zoals de radiobuis, de transistor en tegenwoordig het geïntegreerde circuit. Maar, zoals Dijkstra terecht stelt, de mogelijkheden van de techniek overtreffen op dit ogenblik verre ons vermogen om het werken ermee te beheersen.

Met computers kunnen veel werkzaamheden sneller worden uitgevoerd en kunnen problemen worden opgelost die tevoren onoplosbaar leken. Wij hebben daartoe hulpmiddelen ontwikkeld zoals programmeertalen, waarin oplossingsmethodes worden geformuleerd en die onze machines moeten besturen. Veel van die hulpmiddelen bleken echter onvoldoende ten opzichte van de steeds ingewikkelder problemen en oplossingen. Software-ontwikkeling bleek niet arbeidsbesparend maar uiterst arbeids-intensief: de *softwarecrisis* was een feit.

1.1 Het Probleemgebied



In het volgende hoofdstuk zullen wij nader ingaan op het feit dat de symptomen van de softwarecrisis "optreden in de vorm van software die niet beantwoordt aan de behoeften van de gebruiker, overmatig duur is, moeilijk aan te passen is, moeilijk onderhoudbaar is en die niet opnieuw te gebruiken is" [1]. Gedurende de laatste tien, twintig jaar zijn tal van software-ontwikkelingsprojecten helemaal of bijna helemaal mislukt. Het is pas sinds kort dat wij enigszins beginnen te begrijpen wat de moeilijkheden zijn bij het ontwikkelen van grote systemen en hoe wij van die ontwikkeling een beheersbaar proces kunnen maken. Software-ontwikkelingsmethoden, bestaande uit veel kunst en vliegwerk zijn al veel te lang toegepast!

Om de problemen van de crisis het hoofd te bieden moet de software-ontwikkelingsproblematiek gedisciplineerd benaderd worden met behulp van goede software-ontwerpmethodes. Alleen een methode is echter niet voldoende: wij moeten ook over een geschikte taal beschikken, waarin het ontwerp kan worden beschreven en waarmee het ontwerp kan worden uitgevoerd.

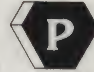
De tot nu toe meest gebruikte taken, FORTRAN en COBOL, werden ontwikkeld in de steentijd van de geschiedenis van de informatica, lang voordat men enig inzicht had in de problematiek van het ontwikkelen van grote systemen. Toen dat inzicht wel begon te dagen heeft men aan dit soort talen allerlei hulpmiddelen, zoals preprocessors, uitbreidingen en systemen voor de administratie van het ontwikkelingsproces, toegevoegd om ze op deze wijze aan te passen aan de nieuwere ideeën. Eigenlijk beperken deze talen onze manier van denken over oplossingsmethodes voor problemen tot methodes die bestaan uit een aantal achtereenvolgens uit te voeren elementaire opdrachten; in dit verband wordt wel gesproken over de *von Neumann mind-set*.

In de tijd waarin deze oude talen werden ontworpen waren de problemen eenvoudig vergeleken met de problemen die men tegenwoordig wil oplossen. FORTRAN werd gemaakt voor wetenschappelijke toepassingen en COBOL voor administratieve toepassingen; voor die beide toepassingsgebieden zijn zij nog steeds redelijk geschikt. Inmiddels heeft zich echter een nieuw groot toepassingsgebied ontwikkeld: dat van de ingebouwde computersystemen.

Een *ingebouwde computer* is een computer die een onderdeel vormt van een groter systeem zoals bijvoorbeeld een geleid projectiel, een industrieel produktieproces, een communicatienetwerk of zelfs een auto of een elektronische oven. De toepassingen binnen dit probleemgebied zijn legio en vaak zeer verschillend. De ingebouwde computer kan in omvang variëren van één microprocessor tot een netwerk bestaande uit een aantal grote computers. Er zijn ook overeenkomsten: de systemen zijn meestal groot, er is sprake van parallelle verwerking, tijdafhankelijke randvoorwaarden en de betrouwbaarheid dient groot te zijn.

Noch FORTRAN, noch COBOL (en ook de meeste andere programmeertalen niet), werd ontworpen voor dit toepassingsgebied. Toch worden er nog steeds systemen voor bijvoorbeeld procesbesturing in COBOL of FORTRAN ontwikkeld, waarbij honderdduizenden regels programmatekst moeten worden geschreven. Het is geen wonder dat wij midden in de softwarecrisis zitten: we gebruiken verouderde en voor andere doeleinden ontworpen gereedschappen.

1.2 Ada's Achtergrond





Naar aanleiding van de softwarecrisis nam het Department of Defense (DoD) van de Verenigde Staten het initiatief tot de ontwikkeling van een krachtig ontwerpinstrument: de programmeertaal Ada. Ada werd, in tegenstelling tot andere programmeertalen, ontworpen voor het specifieke probleemgebied van de ingebouwde computersystemen en tevoren werd een programma van eisen opgesteld waaraan de taal zou moeten voldoen. Ada werd niet ontworpen door een comité, maar door een klein ontwerpteam. Daarna werden verfijningen en verbeteringen aangebracht op grond van uitgebreide en openbare beoordelingen door deskundigen. In hoofdstuk 3 zal nader worden ingegaan op de wijze waarop dit ontwikkelingsproces plaatsvond.

Ada is een "belangrijke stap vooruit op het gebied van programmeertechnieken en combineert de belangrijkste gedachten op dit gebied op een zo systematische wijze dat aan de behoeften in de programmeerpraktijk tegemoet wordt gekomen" [2]. Dit wil niet zeggen dat er geen kritiek is geweest op Ada. De taal zou te ingewikkeld zijn en daarom niet praktisch toepasbaar. Hiermee zijn wij het volstrekt niet eens: Ada is gebaseerd op een kleine verzameling van eenvoudige begrippen, zoals abstracte gegevensvoorstelling, beperkt toegankelijk maken van gegevens en exacte specificatie van datatypen.

Ada is een taal, waarin veel moderne ideeën over ontwerpen van systemen zijn opgenomen en is daarom bij uitstek geschikt om er probleemoplossingen in te formuleren. Ada moedigt niet alleen aan tot het gebruik van goede ontwerp- en programmeermethodes, maar dwingt daar zelfs toe, zoals we in de volgende hoofdstukken zullen zien. Net als andere instrumenten die een omwenteling teweeg brachten, helpt Ada ons bij het formuleren van nieuwe en vaak veel efficiëntere oplossingen.

Behalve specificaties voor de taal heeft het DoD ook specificaties opgesteld voor een programmeeromgeving, de *Ada Programming Support Environment* (APSE). Het doel van APSE is om alle aspecten van het ontwikkelen van programmatuur in Ada te ondersteunen. In hoofdstuk 22 zullen deze specificaties in detail worden bekeken. De taal Ada, de programmeeromgeving APSE tesamen met wat men zou kunnen noemen een Ada geesteshouding, vormen de *Ada-cultuur*, waarmee onze programmeerproblemen beheersbaar kunnen worden.

1.3 Ada's Invloed op Systeembouw-technieken




De bekende taalkundige en psycholoog Benjamin Whorf stelt dat talen " een belangrijke invloed kunnen hebben op de wijze van denken, hoewel het niet zo is dat deze volledig door de taal bepaald wordt" [3]. De Ada-cultuur zou dus diezelfde invloed kunnen hebben. Met Ada kan de von Neumann denkwijze doorbroken worden en kunnen oplossingen veel meer in termen van het probleem waar het om gaat geformuleerd worden. De oplossingen worden daardoor leesbaarder, betrouwbaarder en gemakkelijker te onderhouden. Met behulp van Ada kunnen nu problemen worden opgelost, die tevoren te complex leken om een correcte oplossingsmethode te kunnen garanderen.

"Programmeertalen zijn noch de oorzaak, noch de oplossing van onze programmeerproblemen, maar omdat zij een belangrijke rol spelen bij het software-ontwikkelingsproces kunnen zij ofwel de bestaande problemen nog groter maken, ofwel de oplossing daarvan vereenvoudigen" [4]. Wat Ada betreft mag men het laatste verwachten. Ada kan de softwarecrisis niet oplossen, maar tesamen met goede ontwerpmethoden beschikken wij over een machtig gereedschap om ons daarbij te helpen.

2 DE SOFTWARECRISIS

Het ontwerpen van softwaresystemen vergt veel denkwerk. Het op tijd afleveren van een efficiënt, betrouwbaar en eenvoudig te onderhouden systeem is een nog gigantischer taak, zeker als het gaat om grote procesbesturende systemen. De vakman die een dergelijk systeem creëert is zowel een wetenschapsman als een kunstenaar.

Enerzijds is de programmeur een wetenschapsman of -vrouw, die uitgaat van een formele theorie en een verzameling vaste regels. Tijdens het programmeren kunnen bijvoorbeeld technieken als gestructureerde analyse en object-gerichte ontwerpmethodes worden toegepast, of kan gebruik worden gemaakt van resultaten uit de wachtrijtheorie of de numerieke analyse. Anderzijds is de programmeur een kunstenaar, die de onderdelen van een systeem als het ware boetseert uit het ruwe materiaal van datastructuren en algoritmen en die daarna de onderdelen tot een geheel samenvoegt. Het is deze tweeslachtigheid die de computerwetenschap tot een uitdaging maakt, maar die ook de oorzaak is van veel problemen.

Het gaat hier om een betrekkelijk nieuwe wetenschap en tot nu toe is het nog niet gelukt er een algemeen aanvaard systeem van theorie en kennis van te maken [1]. Te vaak nog wordt alleen op artistieke creativiteit vertrouwd om programmeerproblemen tot een oplossing te brengen. Toch lukt het meestal niet met behulp van (tover)kunsten complexe problemen op te lossen; nog steeds bevinden wij ons midden in de softwarecrisis.

2.1 Softwarecrisis: Wat Houdt Het In?



Te constateren dat wij in een softwarecrisis zitten is bijna een cliché [2]. In ieder geval hebben we er al heel lang mee leren leven: vanaf die goede of kwade dag toen Augusta Ada Lovelace haar ganzeveder op het papier zette om programma's voor de 'analytic engine' van Charles Babbage te schrijven. Toch heeft het geduurd tot de internationale conferentie over software engineering in Garmisch, West Duitsland, in 1968, voordat het bestaan van de crisis in het openbaar bekend werd [3].

Eigenlijk komt de hele softwarecrisis hier op neer, dat het veel moeilijker blijkt softwaresystemen te bouwen dan dat men op het eerste gezicht zou denken. Je zou toch zeggen dat het er alleen maar om gaat wat symbolen op een rijtje te zetten die de computer vertellen wat hij moet doen? De praktijk leert echter dat zo'n simpel wereldbeeld niet erg realistisch is.

Tijdens het onderwijs in de informatica worden systemen gebouwd die bestaan uit hoogstens een paar honderd programmaregels en in dat geval is het hele systeem gemakkelijk te overzien. Ook het wijzigen van dergelijke programmatuur is niet zo moeilijk, omdat de programmeur meestal een paar dagen later nog wel weet hoe zijn programma is opgebouwd. Als tijdens het testen heel erg grote problemen ontstaan kan het hele systeem zelfs vrij eenvoudig opnieuw ontworpen worden.

Als het gaat om het ontwerpen en bouwen van een systeem dat bestaat uit tienduizenden of zelfs miljoenen programmaregels, dan hebben we te maken met een heel ander probleem. Zo'n systeem kan onmogelijk door één man ontworpen of gebouwd worden. Door meer mensen op hetzelfde project te zetten krijgen we communicatie- en coördinatieproblemen die onze zorgen alleen maar groter maken. Het veranderen van een dergelijk systeem is moeilijk, want meestal is er niet één man of vrouw die het hele systeem volledig kan overzien. Het systeem moet daarom met behulp van documentatierapporten beschreven worden; een moeizaam proces dat meestal weken achterloopt op de werkelijke ontwikkeling. En als bij het testen problemen opduiken dan is meestal noch de tijd, noch het geld beschikbaar om het hele systeem opnieuw te ontwerpen.

Elke programmeur die aan een groot project heeft meegewerkt heeft zijn of haar vreugde in het creatief bezig zijn wel eens voelen omslaan in wanhoop en frustatie. Als je dan vraagt wat het probleem nu eigenlijk was komen er antwoorden als: "die module had onverwachte bij-effecten" of "het interface was niet correct gedefinieerd". We zullen zien dat dit maar symptomen zijn van veel dieper liggende problemen. Meestal leiden al die symptomen bij elkaar tot systemen die niet op tijd afkomen, die duur zijn, onbetrouwbaar en die vaak niet voldoen aan hun specificaties. Dat zijn dan de direct zichtbare gevolgen van de softwarecrisis.

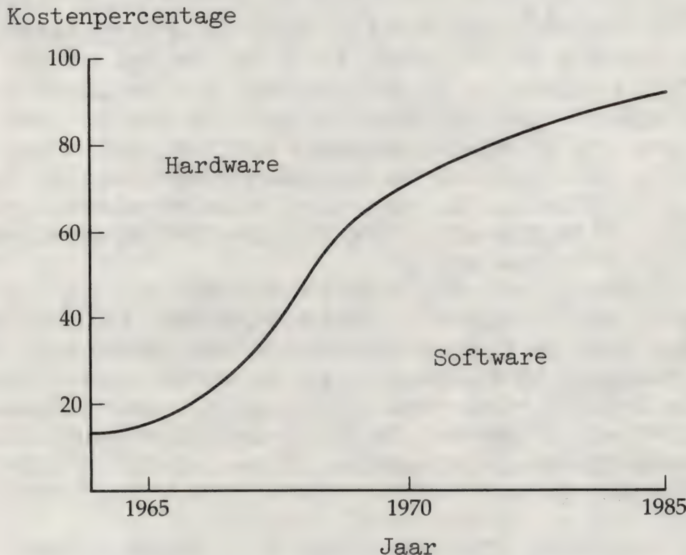
Intuïtief zien we wel dat er problemen zijn, maar het is niet zo eenvoudig om hun gevolgen volledig te overzien. Zoals David Fisher stelde: "Hoewel er veel algemeen erkende symptomen zijn, zijn hun oorzaken niet goed omschreven en zijn er weinig hanteerbare maatstaven voor het beoordelen van het belang van de waargenomen problemen en van de doelmatigheid van de voorgestelde oplossingen" [4]. Hij vervolgt met het opsommen van een aantal symptomen van de softwarecrisis [5]:

- *Functionaliteit*. Geautomatiseerde systemen voorzien vaak niet in de behoeften van de gebruiker.
- *Betrouwbaarheid*. Software vertoont vaak fouten.

- *Kosten*. De kosten zijn vaak onvoorspelbaar en vaak onnodig hoog.
- *Flexibiliteit*. Onderhoud is vaak moeilijk, kostbaar en kan leiden tot nieuwe fouten.
- *Levertijd*. Software wordt vaak te laat opgeleverd en kan vaak minder dan voorgespiegeld was.
- *Overdraagbaarheid*. Software die op het ene systeem draait kan zelden op een ander systeem worden overgezet, ook al zou het precies dezelfde functies moeten verrichten.
- *Doelmatigheid*. Tijdens de ontwikkeling van software wordt geen optimaal gebruik gemaakt van de beschikbare middelen, met als doel korte verwerkingstijden en spaarzaam geheugengebruik.

Software is een moeilijk te vatten produkt: je kunt het niet aanraken en je kunt er maar weinig van zien. Dat is misschien ook de reden dat zo weinig van onze symptomen echt meetbaar zijn te maken. We begrijpen nog niet helemaal wat het is dat we willen meten, hoewel het vakgebied van de 'programmetrie' al een paar antwoorden op onze vragen heeft geproduceerd. Toch zijn er nog weinig statistische gegevens, die kunnen helpen een inzicht te geven in de omvang van de softwarecrisis.

Uit figuur 2-1 is op te maken dat de verhouding tussen hardware- en softwarekosten bij het DoD zich snel heeft ontwikkeld naar



Figuur 2-1 Verloop van de hardware/software kostenverhouding.

het punt waar de softwarekosten de hardwarekosten overtreffen [6]. Wat misschien nog erger is: de kosten voor software-onderhoud zijn vaak hoger dan die van de oorspronkelijke ontwikkeling. Hoewel verschillende rapporten verschillende cijfers noemen, geeft de industrie tussen de 40% en 75% van het totaal bedrag voor hardware en software uit aan software-onderhoud. En hier gaat het dan nog om percentages: de totale softwarekosten zijn enorm aan het toenemen, terwijl computers steeds goedkoper worden en terwijl er steeds meer probleemgebieden worden ontdekt waar automatisering kan worden toegepast. In het begin van de zeventiger jaren waren de totale softwarekosten voor het DoD meer dan drie miljard dollar; een onderzoek van de Electronics Industries Association (EIA) voorspelt dat de totale softwarekosten voor alleen ingebouwde computersystemen in 1990 de 32 miljard dollar zullen overtreffen.

Natuurlijk blijken onze softwareproblemen niet alleen uit de kostenbedragen. We hoeven alleen maar te wijzen op de vele voorbeelden van grote softwareprojecten, die achterraakten op schema, of werden voltooid terwijl maar een klein deel van de oorspronkelijke plannen gerealiseerd werd. De twee bekendste voorbeelden zijn wel het IBM 360 operating system en het World Wide Military Command and Control System (WWMCCS), maar we zouden nog tal van andere voorbeelden kunnen noemen van projecten in heden en verleden, waarbij men in de problemen is geraakt. Men mag verwachten dat dit zo zal blijven, zolang men blijft volharden in het gebruik van verouderde methodes en gereedschappen.

2.2 Wat Zijn De Oorzaken Van De Crisis?





Tot nu toe hebben we alleen gesproken over de symptomen van de softwarecrisis. Om het probleem te kunnen aanpakken moeten we eerst iets over de oorzaken weten. In het voorafgaande werd de ontwerper van software met een kunstenaar vergeleken, maar als alleen maar op artistieke creativiteit vertrouwd wordt dan zijn de resultaten vaak onvoldoende, zoals we ook eerder zagen. Dit betekent natuurlijk niet dat creativiteit bij het programmeren geen rol speelt: het is juist de vonk van de creativiteit die tot een originele oplossingsmethode kan leiden. Problemen ontstaan pas als het creatieve proces niet beheerst wordt. Een concert ontaardt dan in een kakofonie; een programma in een duur, onbetrouwbaar en niet te onderhouden produkt.

M.T. Devlin geeft een aantal redenen waarom men toch vaak vertrouwt op ongebreideld creatief vermogen [7]:

- Organisaties blijken vaak niet in staat de effecten van de softwarelevenscyclus op hun juiste waarde te schatten.

- Er is een tekort aan goed opgeleide software-ontwikkelaars.
- De von Neumann architectuur van de meeste computers bemoeilijkt het toepassen van moderne programmeermethodes.
- Organisaties blijven vaak vasthouden aan verouderde programmeertalen en programmeermethodes.

Dit laatste wordt nog eens benadrukt door de opmerking dat "geen enkele manager nog een eerste-generatie buizencomputer zou willen hebben, maar dat maar weinigen afstand doen van eerste-generatie programmeertalen, zoals FORTRAN" [8].

We hebben te maken met complexe problemen en toch, zoals we in hoofdstuk 4 nog zullen zien, zijn de huidige softwaregereedschappen niet in staat ons te helpen bij het beheersen van de complexiteit van onze oplossingen. Onze problemen zullen niet eenvoudiger worden, want als onze gereedschappen beter worden zullen steeds nieuwe en ingewikkelder problemen oplosbaar worden.

Het feit dat wij niet in staat zijn de complexiteit van onze oplossingen te beheersen is de uiteindelijke oorzaak van de softwarecrisis. W.A. Wulf vat dit als volgt samen: "Het zijn onze menselijke beperkingen, ons onvermogen om alle relaties en implicaties in een complexe situatie tegelijkertijd te overzien, waardoor de sofwarecrisis in essentie wordt veroorzaakt" [9]. Al in 1965 raakte Dijkstra de kern van dit probleem toen hij zei: "Ik heb maar een heel klein hoofd en daar moet ik mee leren leven" [10].

2.3 Wat Kunnen We Tegen De Crisis Doen?





Men zou kunnen denken dat ons menselijk onvermogen het onmogelijk maakt dat wij ooit met succes oplossingen creëren voor ingewikkelde softwareproblemen. Toch is dit niet het geval: het graven van een kuil door één man met zijn blote handen is moeilijk; het Panamakanaal zou hij zeker niet op die manier kunnen graven. Dat het kanaal toch werd gegraven komt omdat gebruik werd gemaakt van krachtige gereedschappen. Ditzelfde geldt bij software-ontwikkeling: we moeten gebruik maken van gereedschappen die ons helpen het proces van het ontwikkelen van complexe systemen te beheersen.

Dergelijke softwaregereedschappen zijn onder andere: technieken voor gestructureerd programmeren, gegevensstroomdiagrammen en object-georiënteerde ontwerpmethodes. Al deze instrumenten staan in direct verband met een aantal basisprincipes voor software-ontwikkeling, zoals wij in het volgende pakket zullen laten zien. Zonder toepassing van modernere programmeermethodes blijkt de produktiviteit betrekkelijk constant te zijn (ongeveer tien regels correcte programmatekst per dag), onafhankelijk van de gebruikte programmeertaal [11]. Dit leidt tot de conclusie dat hogere

programmeertalen gebruikt moeten worden voor de formulering van oplossingen, omdat iedere instructie daarin veel krachtiger is dan in assembleertaal. Toch moet de programmeertaal met zorg gekozen worden, want de taal heeft grote invloed op het uiteindelijke ontwerp van de oplossing. Dit is ook waar wij op doelden toen wij spraken van een denkmethode die aan onze manier van denken bepaalde beperkingen oplegt.

De uiteindelijke oplossing voor het probleem dat de oorzaak is van de softwarecrisis, namelijk de beperkte mogelijkheden van de mens, kan gevonden worden in de toepassing van moderne software-ontwikkelingsmethodes, ondersteund door een hogere programmeertaal die deze methodes aanmoedigt en ondersteunt. In het volgende hoofdstuk wordt verder ingegaan op de ontwikkeling van zo'n hogere programmeertaal.

Oefeningen

1. Bekijk het grootste softwaresysteem dat u, ofwel alleen, ofwel als lid van een team ontwikkeld heeft. Bereken het gemiddelde aantal regels programmatekst per persoon per dag. Hoe dicht ligt dit aantal bij 10 regels per dag?

2. Ons vermogen om complexe situaties te kunnen overzien hangt samen met onze geheugencapaciteit. Geef enkele voorbeelden die onze beperkingen in dit opzicht illustreren (denk bijvoorbeeld aan het onthouden van telefoonnummers van meer dan 9 cijfers).

3. *Waar of niet waar?* Hoe later een fout wordt ontdekt tijdens de ontwikkeling van een softwaresysteem, des te duurder is het die te herstellen. Licht uw antwoord toe.

*4. Misschien wel het ernstigste symptoom van de softwarecrisis is de grote hoeveelheid software van inferieure kwaliteit die wordt geproduceerd. Wat is volgens u het verschil tussen 'goede' en 'slechte' software?

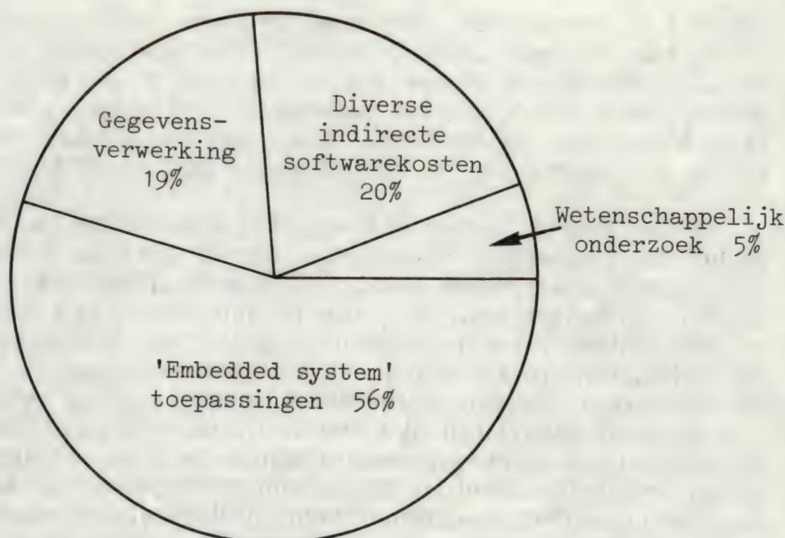
3 ADA'S ONTWIKKELINGSGESCHIEDENIS

Wij zitten op het ogenblik gevangen in de softwarecrisis. Er zijn tal van suggesties gedaan over de wijze waarop wij de crisis zouden kunnen bestrijden en in het volgende pakket zullen wij enkele van de vrij algemeen aanvaarde ontwikkelingsmethodes nader bekijken. In ieder geval moeten wij, welke ontwerpmethode wij ook kiezen, een geschikte programmeertaal hebben om onze oplossingen te formuleren. Hoewel in theorie elke berekening die in de ene programmeertaal kan worden geformuleerd ook in elke andere programmeertaak kan worden geschreven zullen de uiteindelijke oplossingen vaak zeer verschillend zijn en zullen sommige oplossingen veel begrijpelijker zijn dan andere [1].

Een programmeertaal geeft vorm aan de manier waarop wij denken over de oplossingen voor onze problemen. Ideaal zou een programmeertaal zijn, waarin systemen kunnen worden geformuleerd op een wijze die direct aansluit bij de op te lossen problemen, en waarin ook ingewikkelde systemen overzichtelijk blijven. Zo'n taal is Ada. Ada is niet het laatste woord dat op het gebied van programmeertalen gesproken zal worden (als dat al ooit gesproken wordt), maar Ada is geschikt voor het maken van grote, betrouwbare en onderhoudbare systemen. Natuurlijk is Ada niet uit het niets ontwikkeld, er is een voorgeschiedenis aan vooraf gegaan. Deze voorgeschiedenis zullen wij in dit hoofdstuk beschrijven en verklaren.

3.1 De Fase Van De Probleemanalyse  

In het begin van de jaren zeventig signaleerde het departement van defensie van de Verenigde Staten dat de softwarekosten voor grote systemen bleven toenemen. Voor die tijd waren het steeds de hardwarekosten geweest die de softwarekosten verre overtroffen. Welke problemen nu precies een rol speelden bij het ontwikkelen van grote systemen wist men niet, maar de effecten in de vorm van slecht management, late oplevering, onbetrouwbare programmatuur en zeer hoge kosten werden wel degelijk gevoeld. In 1973 besloegen softwarekosten ongeveer 46% (meer dan drie miljard dollar) van het



Figuur 3-1 Geschatte softwarekosten bij DoD in 1973.

totaal van de computerkosten van $7.5 miljard bij het DoD. In figuur 3-1 is te zien dat 56% van deze softwarekosten de kosten voor ingebouwde systemen betreft. Administratieve toepassingen (hoofdzakelijk in COBOL) besloegen 19% en wetenschappelijke toepassingen (meestal in FORTRAN) maar 5% [2]. De overige 20% bestond uit diverse kosten en indirecte kosten.

Tussen 1968 en 1973 namen de directe kosten van de geautomatiseerde systemen bij het DoD toe met 51%, terwijl de hardwarekosten in die tijd toch zeer sterk daalden [3]. De stijgende kosten waren maar een deel van het probleem, waarmee het DoD te kampen had. Het zou mogelijk geweest zijn dat wij gezegd hadden: "Nu ja, het is duur, maar we krijgen dan ook de kwaliteit die we verlangden" [4]. In werkelijkheid, en dat zagen we al in hoofdstuk 2, kwam het maar zelden voor dat er kwaliteit geleverd werd.

Daar kwam nog bij dat ons softwaregereedschap in een aantal opzichten tekort schoot:

- Er bestonden te veel verschillende programmeertalen.
- Er werden talen gebruikt die niet geschikt waren voor de toepassingen die erin geprogrammeerd werden.
- Er bestonden geen geschikte programmeeromgevingen.

Er bestonden in die tijd minstens 450 verschillende programmeertalen voor algemene toepassingen. Het kunnen er zelfs 500 tot 1500 geweest zijn, afhankelijk van de bron die men raadpleegt [5].

Omdat het gebruik van talen door DoD niet centraal gecontroleerd werd, kon iedereen voor elk project zijn eigen taal ontwikkelen, of een of ander vreem dialect van een bestaande taal gebruiken [6]. Zo moesten telkens opnieuw mensen worden opgeleid in een nieuwe taal, was er nauwelijks sprake van enige uitwisseling van kennis tussen projectgroepen en werden geldmiddelen ineffectief gebruikt [7].

Vaak zaten projecten vast aan één leverancier of aan verouderde technieken; zo werd herhaaldelijk COBOL gebruikt voor procesbesturingstoepassingen en werd assembleertaal gebruikt voor administratieve toepassingen. Veel van de gebruikte talen ondersteunden op geen enkele wijze de modernere gedachten over programmeermethodes. Omstreeks 1970 werden de ideeën achter de term gestructureerd programmeren nauwelijks begrepen en nog minder aanvaard.

De grote diversiteit in gebruikte talen bracht de mensen bij DoD er ook niet toe softwaregereedschappen te gaan ontwikkelen. Meestal waren slechts de absoluut noodzakelijke hulpmiddelen beschikbaar, zoals een compiler, een linker voor bibliotheekprocedures en een loader. Syntax-georiënteerde editors of een instrumentarium voor configuratie management waren (en zijn) zaken waarvan men hoogstens wel eens gehoord had; men had het veel te druk met het worstelen met de softwarematerie om aan dergelijke academische zaken aandacht te kunnen besteden.

Hoewel toepassingen van ingebouwde computersystemen bij DoD vaak zeer verschillend zijn, hebben zij toch een aantal gemeenschappelijke eigenschappen, namelijk [8]:

- *Omvang*. Duizenden tot miljoenen regels programmatekst.
- *Lange levensduur*. Tien tot vijftien jaar.
- *Voortdurend onderhevig aan wijzigingen*. Wegens veranderende specificaties.
- *Technische randvoorwaarden*. In apparatuur, geheugenruimte en verwerkingssnelheid.
- *Hoge graad van betrouwbaarheid*. Tevens een grote robuustheid (op kunnen vangen van fouten).

Het grootste deel van de softwarekosten bij het DoD betrof ingebouwde ('embedded') computersystemen en dit is de reden dat DoD daar in de eerste plaats zijn aandacht op richtte. Een ingebouwd computersysteem kan worden omschreven als een subsysteem van een groter systeem, waarbij het niet in de eerste plaats gaat om rekenresultaten, maar om het besturen van bijvoorbeeld een raket of om het beheersen van een proces. Zo'n 'embedded' systeem kan variëren van een systeem bestuurd door één microprocessor tot een systeem bestaande uit een netwerk van een aantal grote computers. Deze ingebouwde systemen stellen hun eigen specifieke eisen, zoals:

- Parallelle verwerking
- Van reële tijdparameters afhankelijke besturing (real-time control)
- Vermogen tot opvangen van bijzondere situaties (exception handling)
- Speciale eisen voor gegevensinvoer en gegevensweergave (I/O control).

Langzamerhand begon het DoD in te zien dat een hogere programmeertaal uiteindelijke voor de totale systeemlevenscyclus kostenbesparend zou kunnen werken. De apparatuur werd trouwens sneller en betrouwbaarder en men wist nu zoveel van compilerbouw dat voldoende efficiënte vertalingen van in een hogere programmeertaal geformuleerde systemen voor real-time toepassingen mogelijk leken. Een geschikte programmeertaal bestond echter nog niet en daarom stelden in 1974 het leger, de marine én de luchtmacht onafhankelijk van elkaar voor, een dergelijke taal te gaan ontwikkelen.

In januari 1975 stelde Malcolm Currie, directeur van het defensie researchcentrum voor om gezamenlijk één geschikte taal te gaan ontwikkelen. Daartoe werd een werkgroep opgericht, HOLWG genaamd (High Order Language Working Group). In de HOLWG zaten vertegenwoordigers van alle onderdelen en er werd samengewerkt met afgevaardigden uit Engeland, West Duitsland en Frankrijk. De HOLWG had als opdracht [9]:

- Specificaties vast te stellen voor DoD hogere programmeertalen.
- Bestaande talen op grond van die specificaties te beoordelen.
- Een aanbeveling te doen voor een basisset van programmeertalen.

Opdat de pogingen van de HOLWG niet zouden verwateren werd de ondersteuning van de ontwikkeling van nieuwe talen tijdelijk geheel stopgezet.

3.2 De Specificatiefase





De HOLWG had als eerste opdracht specificaties voor bij het DoD te gebruiken talen te formuleren. In april 1975 werd het STRAWMAN [10] rapport gepubliceerd en voor commentaar toegezonden aan legervertegenwoordigers, de industrie en de academische gemeenschap. Er werd ook om commentaar gevraagd aan experts in de Europese informaticawereld.

Op grond van de reacties op STRAWMAN, werd WOODENMAN [11] geschreven in augustus 1975 en opnieuw in brede kring gepubliceerd. Commentaar hierop leidde tot een volledig programma van eisen, TINMAN [12] genoemd, gepubliceerd in januari 1976. Dit rapport omschreef alle eisen waaraan een hogere programmeertaal volgens het DoD zou moeten voldoen.

De eerste versie van elk van de genoemde documenten werd geschreven door David Fisher, met belangrijke bijdragen van P.R. Wetherall. Aan de uiteindelijke versies droegen echter meer dan 200 personen uit 85 DoD organisaties, 26 industrieën, 16 universiteiten en 7 andere organisaties bij [13].

Op het eerste gezicht zou men denken dat deelname van zoveel mensen tot een verwarde verzameling van elkaar tegensprekende specificaties moet hebben geleid. Merkwaardig genoeg kwamen alle specificaties wonderwel overeen, zelfs als het heel verschillende toepassingsgebieden ging.

Bijna elke groep wenste een taal die begrippen zou ondersteunen als datastructuren, data-abstractie en het ontoegankelijk maken van bepaalde gegevens ('information hiding'). De meeste groepen wilden dat de taal mogelijkheden zou bevatten voor real-time besturing en voor exception handling (het correct opvangen van uitzonderlijke situaties). Het zag er weliswaar nog niet naar uit dat één enkele taal aan alle geformuleerde wensen zou kunnen voldoen, maar áls zo'n taal zou bestaan zou deze zeker in een behoefte voorzien.

In oktober 1976 organiseerde de HOLWG aan de Cornell universiteit een internationale workshop om de TINMAN specificaties te bespreken. Tegelijkertijd werden er voorbereidingen getroffen voor het introduceren van één taal bij het DoD.

In april 1976 werd DoD rapport 5000.29 gepubliceerd, getiteld: "De beheersing van computertoepassingen in grotere verdedigingssystemen" [14]. Dit rapport schreef het gebruik voor van een door het DoD goedgekeurde hogere programmeertaal voor systemen op defensiegebied, tenzij men kon aantonen dat het gebruik van een andere taal tot aanmerkelijk lagere kosten zou leiden. Gevolg van dit voorschrift was dat het gebruik van assembleertalen duidelijk afnam, en dat vaker voor een hogere programmeertaal werd gekozen, die al eerder met succes was toegepast. Door alleen goedgekeurde talen toe te staan, zag het er naar uit dat het DoD eindelijk de programmeertaal wildgroei begon af te remmen.

Tegelijkertijd begon de HOLWG met een formele beoordeling van bestaande talen op grond van de TINMAN-specificaties. In 1976 werd een lijst opgesteld van goedgekeurde programmeertalen (DoD rapport 5000.31 [15] van november 1976). Op de lijst stonden de volgende talen:

- Dod — FORTRAN, COBOL
- Landmacht — TACPOL

- Marine CMS-2
  SPL/1
- Luchtmacht JOVIAL J3
  JOVIAL J73

In januari 1977 was de beoordeling van de bestaande talen op grond van de TINMAN-specificaties voltooid [16]. Het werd een document van 2800 pagina's. Er werden 23 verschillende talen onderzocht, waaronder FORTRAN, COBOL, PL/1, HAL/S, TACPOL, CMS-2, SPL/1, JOVIAL J3, JOVIAL J73, ALGOL 60, ALGOL 68, CORAL 66, Pascal, SIMULA 67, LIS, LTR, RTL/2, EUCLID, PDL2, PEARL, MORAL en EL/1. Er werd ook nog een aantal andere talen bekeken om na te gaan of die nog speciale eigenschappen hadden, die in de uiteindelijke taal zouden moeten worden opgenomen. De conclusie luidde [17]:

- Er bestond geen geschikte taal voor DoD's embedded computertoepassingen.
- Het was wenselijk dat één enkele taal gebruikt zou worden.
- Het ontwikkelen van een dergelijke taal was mogelijk.
- De nieuwe taal zou een geschikte bestaande taal als uitgangspunt moeten nemen.

De onderzochte talen werden in drie categorieën gerangschikt:

- *Niet geschikt.* Verouderde talen en talen met een ander toepassingsgebied.
- *Niet ongeschikt.* Talen met enkele eigenschappen geschikt voor het probleemgebied.
- *Geschikt.*

Volgens de onderzoekers was geen enkele van de beoordeelde talen geschikt, maar als uitgangspunt zouden Pascal, ALGOL en PL/1 kunnen dienen.

In januari 1977 werd TINMAN opgevolgd door IRONMAN [18], waarin de bevindingen van de onderzoekers waren vastgelegd. Het verschil tussen TINMAN en IRONMAN is eigenlijk niet groot, alleen de indeling is wat anders. Op verzoek van een stuurgroep onder leiding van Barry DeRoze werden in de periode van januari 1977 tot november 1977 twee onafhankelijke onderzoeken gedaan naar de economische haalbaarheid van een taal op basis van de TINMAN specificaties. Beide onderzoeken kwamen tot een positief advies en suggereerden dat het gebruik van één enkele hogere programmeertaal door het DoD tot een besparing van honderden miljoenen dollars per jaar zou kunnen leiden [19].

Op grond van de resultaten van deze rapporten gaf de stuurgroep aan de HOLWG opdracht te beginnen met een ontwikkeling van

een hogere programmeertaal, die in de technische pers de naam DoD-1 kreeg. Het DARPA (Defense Advance Research Projects Agency) kreeg de opdracht met de organisatie te beginnen; projectleider was William Carlson.

3.3 De Ontwerpfase



Het ontwerpen van een programmeertaal is altijd een beetje als een kunst beschouwd. Er bestaan weliswaar basisprincipes, maar er worden ook altijd een aantal subjectieve keuzes gemaakt. Talen als Pascal en SIMULA werden door een kleine groep ontworpen, andere talen, en de bekendste zijn COBOL en PL/1, werden door een commissie ontworpen. Wat DoD-1 betreft was de taak van de ontwerper al enigszins geformaliseerd, omdat de specificaties al vast lagen. Wat het DoD betreft zou de nieuwe taal in ieder geval van kwalitatief hoog gehalte moeten zijn, vooral omdat de taal als algemeen aanvaarde standaard zou moeten kunnen gelden. Het DoD zag overigens ook het belang in van een algemene acceptatie van de taal ook buiten defensiekringen [20].

DoD koos daarom voor een internationale wedstrijd, waarin verschillende groepen ontwerpen voor een taal zouden indienen. Na een beoordeling zouden dan een paar groepen geselecteerd worden om een nader gedetailleerd ontwerp te maken. De hele procedure zou openbaar zijn en zowel het DoD, Amerikaanse industrieën en universiteiten, als Europese industrieën en universiteiten en de overheden van Engeland, Frankrijk en West Duitsland zouden de gelegenheid krijgen, de resultaten te beoordelen. Het DARPA (Defense Advance Research Projects Agency) ging er vanuit dat het op die manier mogelijk moest zijn dat er een taal ontwikkeld werd, gebaseerd op een logische ondergrond en door deskundigen uit de informaticawereld kritisch beoordeeld op kwaliteit.

In april 1977 werd de wedstrijd uitgeschreven in de vorm van een 'Request for Proposal' (RFP), dat wil zeggen een verzoek om voorstellen voor een nieuwe hogere programmeertaal voor algemeen gebruik. Op dit RFP kwamen zeventien reacties en in juli 1977 koos DARPA vier van de deelnemers voor een tweede termijn van zes maanden. Dit was Fase I van de ontwerpprocedure. In die zelfde tijd werd ook een herziene versie van de IRONMAN specificaties gepubliceerd [21]. De vier taalontwerpen uit Fase I waren:

- SofTech (Blue)
- SRI International (Yellow)
- Intermetrics (Red)
- Honeywell/Honeywell Bull (Green)

Opmerkelijk was dat deze talen alle vier Pascal als uitgangspunt hadden gebruikt.

De taalvoorstellen kregen namen van kleuren en werden ontdaan van iedere verwijzing naar hun auteur om een objectieve beoordeling mogelijk te maken. Van februari 1978 tot maart van dat jaar werden de ontwerpen beoordeeld door ongeveer 400 vrijwilligers over de hele wereld in 80 beoordelingsteams. Op grond van de resultaten werden de voorstellen van Intermetrics (Red) en Honeywell (Green) uitgekozen voor de volgende ronde: Fase II.

Tijdens Fase II werd ook alvast een begin gemaakt met het onderzoek naar een programmeeromgeving voor de nieuwe taal. Al in 1977 stelde Whitaker (voorzitter van de High Order Language Working Group HOLWG), dat een programmeertaal alleen niet genoeg zou zijn om de gewenste verbetering in software-ontwikkelingsmethodes te bereiken: de taal zou moeten worden ondersteund met gereedschap van hoge kwaliteit [22].

Begin 1978 verscheen SANDMAN [23], waarin een aantal technische en organisatorische aspecten van programmeeromgevingen werden opgesomd. Naar aanleiding van de reacties op SANDMAN werd door Thoman Standish van de Universiteit van Californië een conferentie georganiseerd om deze aspecten nader te bespreken. De conferentie werd door ongeveer 60 personen uit industrie, leger en universiteiten bijgewoond. Op de conferentie werd onder andere een nieuwe versie van het rapport, PEBBLEMAN genaamd en geschreven door P.F. Elzer [24], besproken. Tegelijkertijd werd opnieuw een versie van de taalspecificatie uitgegeven, STEELMAN [25], waarin een aantal fouten en gebreken die tijdens Fase I waren geconstateerd, verbeterd waren.

In november 1978 was er een openbare vergadering, waarin de ontwerpers van de talen Red en Green technische aspecten van hun voorstellen toelichtten. Tenslotte werd in december 1978 opnieuw een herziene versie van PEBBLEMAN uitgegeven.

In maart 1979 werd Fase II voor Red en Green afgesloten en kon de beoordeling van de talen beginnen. De beoordelingsperiode duurde van maart tot april 1979. Er vonden opnieuw vergaderingen plaats en het moment van de definitieve keuze brak aan.

DoD-1 was nooit officieel aangenomen als naam van de nieuwe taal. Men vond dat te militaristisch klinken en was bang dat dit acceptatie buiten defensiekringen zou bemoeilijken. Lente 1979 bedacht Jack Cooper, een marinemedewerker, de perfecte naam voor de nieuwe taal: Ada, ter ere van Augusta Ada Byron, Countess of Lovelace, en dochter van de dichter Lord Byron [26].

Ada Lovelace (1815-1851) was een wiskundige die samenwerkte met Charles Babbage aan de ontwikkeling van diens rekenmachines: de differentiemachine en later de analytische machine. Zij zag toen al de mogelijkheden van programmeerbare rekenmachines en beschreef hoe dezelfde methode als gebruikt werd voor het specificeren van patronen op het weefgetouw van Jacquard, op Babbage's machine zou kunnen worden toegepast. Zij wordt daarom wel beschouwd als de eerste programmeur ter wereld en na een briefwisseling tussen

het ministerie van defensie en Ada's afstammeling de graaf van Lytton werd officieel toestemming verkregen haar naam te gebruiken [27].

Zo werd in mei 1979 Ada de naam van de nieuwe programmeertaal van het DoD, nadat de taal onder de codenaam Green als winnaar van de wedstrijd was uitgeroepen. In politieke kringen was men verrast dat een Europese mededinger uiteindelijk winnaar werd, maar tijdens de hele procedure had kwaliteit en niet enige politieke overweging als beoordelingscriterium voorop gestaan.

De belangrijkste auteur van Green was de Fransman Jean Ichbiah. Andere teamleden waren de Fransen J. Heliard, O. Roubine en J. Abrial. Verder: P.N. Hilfinger en H.F. Ledgard (Verenigde Staten), J.G.P. Barnes, B.A. Wichmann, M. Woodger en R. Firth (Engeland) en B. Krieg-Bruckner uit Duitsland. In het Ada handboek worden nog een aantal anderen genoemd, die belangrijke bijdragen leverden aan de taal: E. Morel en G. Ferran uit Frankrijk, J.B. Goodenough, M.W. Davis, L. MacLaren, I.R. Nassi, S.A. Schuman en S.L. Vestal uit de Verenigde Staten en I.C. Pyle uit Engeland.

3.4 De Testfase





Met de aankondiging van de winnaar van de wedstrijd was de kous niet af. Integendeel, nu begon Fase III, de fase van het testen en beoordelen van de nieuwe taal. Een ieder werd verzocht een bestaande toepassing te kiezen en die in Ada te programmeren. Als tegenprestatie mochten de vrijwilligers, die hierop ingingen, dan de voorlopige Ada vertaler gebruiken en deelnemen aan één van de vijf cursussen die op verschillende plaatsen in Amerika en Engeland gehouden werden.

Het voorlopige taalhandboek en de verantwoording voor het ontwerp van Ada werden gepubliceerd in één van de tijdschriften van de ACM (Association for Computing Machinery), SIGPLAN Notices, zodat minstens 10.000 mensen er kennis van konden nemen. Het handboek werd ook nog eens aan 2000 daarvoor speciaal uitgezochte deskundigen toegezonden met het verzoek om commentaar. Men wilde op die manier in korte tijd de taal algemeen bekend maken, ontwerpfouten zo snel mogelijk laten signaleren en onduidelijkheden in de taalbeschrijving kunnen corrigeren [28]. Het voorzitterschap van de HOLWG werd inmiddels van Whitaker door Fisher overgenomen.

Bij het begin van fase III gaf de HOLWG een opdracht tot het ontwikkelen van gereedschap ter beoordeling en waardering van compilers, vooruitlopend op het gereedkomen van de eerste Ada compilers. Herfst 1979 werd verder het Ada Configuration Control Board ingesteld, een gezelschap dat wijzigingen in de taal diende te begeleiden en documenteren. Een groep van ervaren beoordelaars moest daarbij van advies dienen.

Oktober 1979 werd een conferentie in Boston gehouden over test- en beoordelingsresultaten en toen in november 1979 Fase III op zijn einde liep waren er meer dan 500 beoordelingen uit 15 landen binnen. Algemene conclusie was, dat het ontwerp acceptabel was, maar dat er nog wat kleine wijzigingen zouden moeten worden aangebracht.

Ook in oktober 1979 verscheen STONEMAN, een nieuwe versie van de specificaties voor een Ada-programmeeromgeving, dit maal geschreven door John Buxton (Harvard Universiteit) en nogmaals herzien in februari 1980 [29].

Op grond van de rapporten uit Fase III, verbeterde het Ada ontwerpteam een aantal tekortkomingen in het eerdere ontwerp. In juli 1980 voltooide Ichbiah het herziene Ada handboek en daarmee kwam een eind aan de periode van ontwerpen en testen.

3.5 De Fase Van Gebruik En Onderhoud





Het wordt nu moeilijk alle ontwikkelingen op het gebied van Ada bij te houden. Velen in militaire, industriële en universitaire kring waren enthousiast over Ada's mogelijkheden, en honderden organisaties begonnen de taal toe te passen, of compilers en APSE's (Ada Programming Support Environments, of programmeeromgevingen) te ontwikkelen. Toch moeten enkele gebeurtenissen apart genoemd worden vanwege hun belang voor de ontwikkelingsgeschiedenis van de taal.

In augustus 1980 keurde de HOLWG het Ada handboek officieel goed. Het werd nu echter wel duidelijk dat de HOLWG het niet alleen meer af zou kunnen. Op 12 december 1980 werd daarom op het eerste ACM symposium over Ada in Boston de AJPO (Ada Joint Program Office) opgericht om alle Ada activiteiten te coördineren [30]. Hoofd werd Larry Druffel en op diezelfde dag werd het document MIL-STD 1815 (Military Standard) goedgekeurd als officiële Ada standaard. De nummering 1815 was trouwens niet toevallig: 12 december 1815 was de datum van de geboorte-aankondiging van Augusta Ada Lovelace, die twee dagen eerder geboren werd.

De AJPO begon al meteen met maatregelen voor de invoering van Ada bij DoD projecten. Men vond dat een wildgroei van uitgebreide en beperkte versies (supersets en subsets) moest worden voorkomen en afwijkende dialecten van Ada werden daarom niet goedgekeurd. In januari 1981 werd om die reden een aanvrage ingediend om Ada als handelsmerk van het DoD te beschermen. Alle militaire instellingen moesten verder beloven dat zij geleidelijk aan Ada als standaardtaal zouden gaan gebruiken, zodat eind 1985 met name voor zogenaamde 'embedded systems' (in een groter systeem ingebouwde computersystemen), geen andere talen meer in gebruik zullen zijn.

Om te voorkomen dat Ada behandeld zou worden als alweer een nieuwe programmeertaal en niet meer dan dat, nam de AJPO het initiatief tot de ontwikkeling van een gids met Ada stijlvoorschriften [31] en een Strategie voor Ada Onderwijs en Opleiding [32]. Op 17 februari 1983, tenslotte, werd de Ada ANSI (Americal National Standards Institute) standaard goedgekeurd.

3.6 Slotwoord





In de ontwikkeling van Ada hebben duizenden computerwetenschapsmensen hun energie gestoken. De lijst van namen is een soort *Wie is Wie* in de computerwereld. De deelname van zo velen had tot een onoverzichtelijk en ingewikkeld resultaat kunnen leiden, maar onder leiding van mensen als Whitaker, Carlson, Fisher, Druffel en Ichbiah is een krachtig en evenwichtig stuk gereedschap geconstrueerd dat ons kan helpen bij een efficiënte ontwikkeling van software systemen.

In het volgende hoofdstuk zullen we de grondslagen van de techniek voor het bouwen van dergelijke systemen behandelen en nagaan hoe Ada ons daarbij kan helpen.

Oefeningen

1. De programmeertaal COBOL werd ook op initiatief van het Amerikaanse departement van defensie (DoD) ontwikkeld. Vergelijk de ontwikkelingsgeschiedenis van COBOL met die van Ada. Wat zijn de verschillen en overeenkomsten? Vergelijk ook de kritiek die beide talen tijdens hun ontwikkeling kregen.

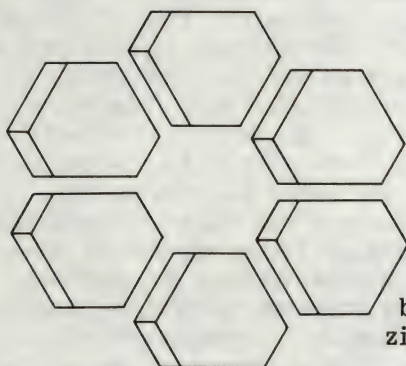
*2. De talen Pascal, Alphard, CLU en LIS hebben alle Ada beïnvloed. Welke invloeden kunt u aanwijzen?

3. Ada is een taal geschikt voor allerlei verschillende toepassingen. Waarom denkt u dat het DoD Ada alleen gepropageerd heeft in het gebied van de embedded systems?

4. Ada is wel "het verlossende woord op programmeertaalgebied" genoemd. Bent u het daarmee eens? Licht uw antwoord toe.

Pakket 2

HIER IS ADA



En de Here zeide: Zie, het is één volk en zij allen hebben één taal. Dit is het begin van hun streven; nu zal niets wat zij denken te doen onuitvoerbaar zijn.

Genesis 11:6

4 SOFTWARE-ONTWIKKELINGSMETHODES

De softwarecrisis wordt veroorzaakt door het verschijnsel dat softwaresystemen steeds ingewikkelder en daardoor moeilijker te beheersen worden. Ook in de toekomst zullen zij zeker niet eenvoudiger worden; immers naarmate ons gereedschap verfijnder wordt en naarmate wij meer ervaring opdoen in het ontwerpen van systemen, kunnen weer nieuwe en complexere klassen van problemen worden aangepakt. We moeten daarom onze toevlucht nemen tot een beheerste ambachtelijkheid, willen wij ooit deze complexiteit de baas kunnen. Dit vak of ambacht wordt *software engineering* genoemd: het ontwerpen en construeren van ingewikkelde programmabouwwerken.

Degenen die van software engineering technieken op de hoogte zijn zullen denken aan gestructureerd programmeren, PDL (Program Design Language of pseudocode), HIPO's (Hierarchical Input Process Output beschrijvingen) en datastroomdiagrammen. Dit zijn enkele van de bekendste hulpmiddelen bij het ontwikkelen van programmatuur. Software engineering methodes helpen ons de hele levenscyclus van softwaresystemen op een consistente wijze te begeleiden. Het is een poging ons werk van 'kunst' tot wetenschap te maken.

In dit hoofdstuk worden de doelstellingen van software engineering aan een nader onderzoek onderworpen en zullen wij enkele elementaire principes behandelen, die ons kunnen helpen die doelstellingen te bereiken. Vervolgens zullen we enige technieken behandelen die deze principes in praktijk brengen. Omdat programmeertalen louter hulpmiddelen zijn om ons ontwerp te formuleren en vervolgens uit te voeren, zullen wij tenslotte een aantal generaties van talen bekijken en nagaan hoe zij onze methodes voor software-ontwikkeling al dan niet ondersteunen. Wanneer we Ada meer in detail gaan bestuderen in dit hoofdstuk, dan zal duidelijk worden dat in Ada deze software-ontwikkelingsprincipes inderdaad in de praktijk zijn gebracht en dat wij met Ada een uitstekend stuk gereedschap tot onze beschikking hebben.

4.1 Doelstellingen Van Software Engineering Technieken





Het meest voor de hand liggende doel bij het ontwerpen van programmatuur is wel dat de gegenereerde oplossing ook de oplossing is van het gestelde probleem. Een probleem op zichzelf is echter dat praktijkproblemen maar heel zelden goed, volledig en op logische wijze gespecificeerd worden. De opdrachtgever, noch de systeembouwer overziet het probleem volledig en meestal worden nadere specificaties pas tijdens de ontwikkeling van het systeem geformuleerd. Het probleem wordt nog gecompliceerder als de apparatuur (hardware) en de programmatuur (software) tegelijkertijd ontwikkeld moeten worden, zoals bij embedded systems vaak het geval is. Verder moeten we er ook rekening mee houden dat de specificaties tijdens de levenscyclus van ons systeem kunnen veranderen; in hoofdstuk 2 wezen we er al op dat het hoogste kostenbedrag zit in de onderhoudsfase. Grote systemen sterven niet; zij worden gewijzigd.

Omdat verandering voortdurend noodzakelijk is tijdens het software-ontwikkelingsproces, zullen doelstellingen geformuleerd moeten worden die daar rekening mee houden. In hun klassieke artikel omschrijven O.T. Ross, J.B. Goodenough en C.A. Irvine deze doelstellingen: "De volgende vier eigenschappen kunnen dienen als doelstellingen bij software-ontwikkeling: wijzigbaarheid, doelmatigheid, betrouwbaarheid en begrijpelijkheid." [1].

Wijzigbaarheid

Wijzigbaarheid is een moeilijk te bereiken en te meten doel. Het gaat hier om "beheerste verandering, waarbij sommige delen of aspecten onveranderd blijven, terwijl andere gewijzigd worden, op een zodanige manier dat het gewenste resultaat bereikt wordt" [2]. Wijzigingen kunnen om twee redenen noodzakelijk zijn: op grond van gewijzigde specificaties, ofwel om een fout te verbeteren.

Bij het ontwerpen van software moet de structuur van het ontwerp op een heldere en logische manier kunnen worden voorgesteld. Omdat de meeste programmeertalen nu niet direct makkelijk leesbaar zijn, moet men vaak zijn toevlucht nemen tot documentatie in de vorm van beschrijvingen en handboeken om die structuur duidelijk te maken. Ideaal zou zijn als de structuur van het ontwerp direct uit de programmatuur zelf zou blijken.

Om problemen bij het wijzigen van programma's te voorkomen dient de bestaande structuur steeds zoveel mogelijk gehandhaafd te blijven. Doet men dat niet, dan ontstaat 'lapwerk' dat de oorspronkelijke structuur te niet doet. Als op die manier een aantal wijzigingen zijn uitgevoerd is de oorspronkelijke structuur volledig verdwenen en wordt het steeds moeilijker nog een wijziging aan te brengen. In een wijzigbaar softwaresysteem is het mogelijk veranderingen aan te brengen zonder dat de complexiteit van het oorspronkelijke systeem toeneemt.

Doelmatigheid

Een softwaresysteem is doelmatig als het de beschikbare middelen zo goed mogelijk aanwendt. De beschikbare middelen betreffen twee factoren: tijd en ruimte. De beschikbare tijd kan een beperking zijn als het proces zich binnen een bepaald tijdsbestek moet afspelen, zoals het uitlezen van meetapparatuur of het reageren op een interruptsignaal. Vanzelfsprekend hangt de verwerkingstijd sterk af van de gebruikte apparatuur, maar ook het gekozen algoritme zal invloed hebben op de verwerkingstijd. Het ruimte-aspect betreft beschikbare geheugenruimte of het aantal beschikbare randapparaten.

Bij toepassingen, waarbij sprake is van een ingebouwd ('embedded') computersysteem, moet vaak zowel met de factor tijd als met de factor ruimte rekening worden gehouden. Als het systeem moet reageren op gebeurtenissen die in de werkelijkheid plaatsvinden, dan is de factor tijd kritiek. Als er aan de hardware beperkingen zijn opgelegd ten aanzien van omvang of energieverbruik, zoals bij een kunstmaan of in een auto, dan kan de factor geheugenruimte een belangrijke rol spelen bij het bepalen van de uiteindelijke oplossing. Vaak is efficiënt gebruik van beide factoren tegelijkertijd niet mogelijk en moet naar een compromis gezocht worden.

In veel gevallen houdt men in een te vroeg stadium van de ontwikkeling al rekening met de efficiëntie van de oplossing. Men concentreert zich op lokale efficiëntie inplaats van op globale efficiëntie. We moeten ervan doordrongen zijn dat "... inzicht in de globale structuur van een probleem vaak een veel grotere invloed heeft op de uiteindelijke doelmatigheid van de oplossing, dan welk gegoochel met bits ook maar mogelijk maakt als wordt uitgegaan van een onjuiste structuur" [3].

Betrouwbaarheid

Bij elk computersysteem dat gedurende lange tijd zonder menselijke tussenkomst moet kunnen werken is de betrouwbaarheid van zeer groot belang. Als zo'n systeem ook nog eens een atoomcentrale of een ruimtevaartuig bestuurt dan zijn de kosten als er iets fout gaat dermate hoog, dat de betrouwbaarheid vrijwel 100% moet zijn. "Betrouwbaarheid betekent zowel voorkomen van fouten in de oplossingsmethode, in het ontwerp en in de architectuur, als ook het opvangen van fouten tijdens de werking" [4].

Als betrouwbaarheid op deze wijze omschreven wordt dan is het duidelijk dat dit criterium tijdens het hele ontwerpproces een rol moet spelen. "Betrouwbaarheid kan alleen vanaf het begin ingebouwd worden; het is niet iets dat nog eens achteraf kan worden toegevoegd" [5]. Honderd procent betrouwbaarheid is echter nooit te bereiken: er zullen altijd onvoorziene omstandigheden mogelijk blijven, zoals een volledig defect raken van een hardware onderdeel, waardoor zelfs het best beveiligde systeem gedeeltelijk vernietigd kan worden. Toch behoort een betrouwbaar systeem zelfs in de meest desastreuze

situaties de schade tot een minimum te beperken (geen kerncentrale-rampen, of uit hun baan rakende satellieten, om bij ons eerdere voorbeeld te blijven).

Begrijpelijkheid

Streven naar begrijpelijkheid kan ons helpen bij het beheersen van complexe softwaresystemen. Begrijpelijkheid is de brug tussen probleemstelling en oplossingsruimte. Dat wil zeggen, om begrijpelijk te zijn moet een systeem op de een of andere wijze een weergave zijn van datgene wat wij in de realiteit waarnemen. Een moeilijk probleem kan alleen succesvol worden opgelost als bij de oplossing gebruik gemaakt wordt van heldere en begrijpelijke structuren. Alleen als die structuur in de software is terug te vinden is het systeem later gemakkelijk te wijzigen, kan het efficiënt werken en is het betrouwbaar.

Er zijn een aantal niveaus aan te geven in de factoren die de begrijpelijkheid van een systeem beïnvloeden. Het laagste niveau is de leesbaarheid door het gebruik van een goede programmeerstijl. Op een hoger niveau moeten zowel de datastructuren (objecten) als de algoritmen (bewerkingen) herkenbaar zijn, die tesamen een afbeelding vormen van de gegevens en processen uit de werkelijkheid. We zullen laten zien dat de begrijpelijkheid sterk afhangt van de programmeertaal die wij gebruiken als uitdrukkingsmiddel voor onze oplossingen.

4.2 Grondslagen Van Software Engineering





De criteria die wij hiervoor formuleerden zijn van toepassing op elke softwaresysteem. Maar het is niet voldoende ze te noemen om vervolgens in het wilde weg te gaan programmeren in de hoop dat er op die manier wel aan voldaan zal worden. Succes is waarschijnlijker als we tijdens het ontwerpen een stel eenvoudige basisprincipes toepassen [6]:

- abstractie
- beperkt toegankelijk maken van gegevens
- modulariteit
- beperking van plaats
- uniformiteit
- volledigheid
- controleerbaarheid

Wij zullen hierna laten zien hoe toepassing van deze principes kan leiden tot gemakkelijk te wijzigen, efficiënte, betrouwbare en begrijpelijke oplossingen.

Abstractie en beperkte toegankelijkheid

Een belangrijk instrument voor het beheersen van complexiteit is *abstractie* [7]. Het begrip abstractie is niet iets nieuws: we passen het toe bij alles wat we doen. Ons programma kan bijvoorbeeld gebruik maken van een schijveneenheid en dit fysieke hulpmiddel kan vanuit een aantal gezichtspunten beschouwd worden:

- Een verzameling logische bestanden.
- Massagegevensopslag met een organisatie in fysieke sporen en sectoren.
- Een verzameling adresseerbare geheugenplaatsen.
- Een fysiek apparaat dat stuursignalen en gegevens verwacht.

We hebben hier een aantal abstractieniveaus aangegeven, waarbij elk niveau is gebaseerd op de onderliggende lagere niveaus. In onze programma's kunnen we steeds dat abstractieniveau kiezen dat past bij ons probleem. In een bestandsbeheerssysteem zien we de schijveneenheid als een verzameling logische bestanden, maar als we als systeemprogrammeur een aansturingsprogramma voor de schijveneenheid moeten schrijven, dan moeten we op het niveau van stuursignalen en datasignalen gaan zitten.

"Bij abstractie gaat het om het vinden van de essentiële eigenschappen, onder weglating van de niet essentiële details" [8]. Als wij een oplossing opdelen in een aantal modules, dan is elk van deze modules een onderdeel van de abstractie op een bepaald niveau. Het principe van abstractie kan zowel worden toegepast op de gegevens als op de algoritmen die een rol spelen in onze oplossing. Zo is in het voorbeeld met de schijveneenheid voornamelijk sprake van gegevensabstractie.

Op ieder abstractieniveau kunnen abstracte datatypen worden gedefinieerd (zoals FILE, RECORD of DISK_SECTOR), ieder bestaande uit een waardeverzameling en een verzameling operaties die op objecten van dat type kunnen worden toegepast. Binnen het databasesysteem moeten bijvoorbeeld verzamelingen van logische gegevensbestanden worden gemanipuleerd en binnen een aansturingsprogramma voor de schijveneenheid wordt de schijveneenheid gezien als een adresseerbaar blok van geheugenlokaties. De schrijver van een databasesysteem is niet geïnteresseerd in fysieke geheugenlokaties op de schijf, terwijl de systeemprogrammeur zich weer niet bezighoudt met logische gegevensbestanden.

Bij het ontwikkelen van algoritmen geldt ditzelfde abstractieprincipe. Ons databasesysteem moet bestanden kunnen openen,

sluiten, lezen en schrijven. Met de problematiek van het selecteren van een spoor op een schijf, het vinden van de juiste sector, het uitvoeren van een controle op correct lezen met behulp van een pariteitensom en het in een buffer zetten van het segment van de gewenste file wensen wij ons dan niet bezig te houden. We zien opdrachten als OPEN, CLOSE, READ en WRITE als abstracte operaties zonder dat het nodig is hun onderliggend mechanisme in detail te kennen. Op deze manier kan het aantal componenten dat een rol speelt voor ons probleem beperkt worden gehouden en kan de verdere uitwerking worden gedelegeerd naar een lager abstractieniveau.

Behalve abstractie hebben wij in dit voorbeeld een tweede basis principe voor software engineering toegepast: het beperkt toegankelijk maken van gegevens, ofwel *information hiding*. De methode van voortschrijdende abstractie is bedoeld om alleen die componenten een rol te laten spelen die essentieel zijn voor het gestelde probleem, "... information hiding is bedoeld om bepaalde details die niet van buiten af door andere componenten van het systeem beïnvloed mogen worden ontoegankelijk te maken" [9]. In het geval van het ontwikkelen van een database betekent het principe van information hiding, dat het systeem zo moet zijn ontworpen dat het voor de applicatieprogrammeur onmogelijk is direct naar een fysieke sector op de schijf te schrijven. Op die manier zou het logische abstractieniveau, waarop sprake is van een logisch gegevensbestand, waarbij fysieke lokaties niet interessant zijn, doorbroken kunnen worden.

'Information hiding' betekent dus het ontoegankelijk maken van het onderliggende mechanisme van een object of een operatie, zodat onze aandacht niet wordt afgeleid van het abstractieniveau waarop we ons bevinden [10]. Het ontoegankelijk maken van het ontwerp op een lager niveau, zoals bijvoorbeeld de fysieke organisatie van bestanden op een schijf, betekent ook dat op een hoger niveau geen gebruik gemaakt kan worden van de implementatiewijze van lagere niveaus. Zouden in ons voorbeeld nieuwe schijveneenheden met een grotere capaciteit worden gebruikt, dan heeft dit, ondanks de benodigde fysieke reorganisatie van de schijf, indien op correcte wijze van het information hiding principe gebruik is gemaakt, geen enkele invloed op het hogere abstractieniveau van logische bestandsstructuren.

Het toepassen van de principes van abstractie en beperkt toegankelijk maken van gegevens heeft invloed op de doelmatigheid van het systeem, maar ook op de andere criteria voor een goed softwareprodukt. De onderhoudbaarheid en begrijpelijkheid van een systeem wordt bevorderd door toepassing van het abstractieprincipe: de programmeur hoeft immers de details van de lagere niveaus niet te kennen. De betrouwbaarheid van het systeem wordt bevorderd als op elk abstractieniveau exact vaststaat welke operaties toegelaten zijn en als voorkomen wordt dat operaties van een lager niveau direct kunnen worden gebruikt.

Modulariteit en lokalisatie

Een volgend belangrijk instrument dat ons kan helpen complexe softwaresystemen te beheersen is *modulariteit*. "Modulariteit heeft te maken met het vereenvoudigen van de weg waarlangs een doelstelling kan worden bereikt. Modulariteit is 'doelmatig structureren'" [11]. Het modulariteitsprincipe wordt in feite steeds intuïtief toegepast, want telkens als een oplossing wordt onderverdeeld in een aantal gedeelten creëren we eigenlijk modules, die in Ada bestaan uit pakketten ('packages'), subprogramma's en taken.

Bij het toepassen van een top-down ontwerpmethode wordt elk volgend niveau gewoonlijk in een aantal verschillende functionele modules onderverdeeld. Modules op het hoogste niveau staan in direct verband met ons abstracte model en zijn dus betrekkelijk machine-onafhankelijk. Modules op een hoog abstractieniveau specificeren *wat* er moet gebeuren en op een lager niveau wordt beschreven *hoe* het moet gebeuren [12]. Gebruiken we een 'bottom-up' ontwerpmethode, dan geldt precies hetzelfde, maar in plaats van het systeem van bovenaf op te delen in deelproblemen, wordt nu het systeem opgebouwd door met elementaire basismodules steeds complexere modules samen te stellen.

Er zijn twee klassen van modules te onderscheiden: functionele (procedure-georiënteerde) modules en declarerende (object-georiënteerde) modules. Een betrouwbaar systeem betekent nauwkeurig omschreven 'interfaces', of communicatiepatronen met en tussen modules. Hoe modules ook worden gekozen, zij zullen zeker communiceren met andere modules; de mate van de benodigde communicatie wordt de mate van *koppeling* genoemd [13]. Hoe losser modules gekoppeld zijn, des te gemakkelijker kan elke module onafhankelijk van de andere behandeld worden. Tegenover koppeling staat *cohesie*, "de mate van samenhang tussen de interne elementen" binnen de module [14]. De cohesie dient zo groot mogelijk te zijn; de componenten binnen een module dienen functioneel samen te hangen.

Toepassen van het principe van *lokalisatie* kan ons helpen bij het ontwikkelen van los gekoppelde, maar coherente modules. Lokalisatie heeft direct te maken met de fysieke afstand tussen objecten [15]. Logisch samenhangende functies dienen dicht bij elkaar, dus bij voorkeur in één module te staan. Dit leidt zowel tot coherente modules als tot los gekoppelde modules.

Modulariteit en lokalisatie ondersteunen onze doelstellingen: wijzigbaarheid, betrouwbaarheid en begrijpelijkheid. Binnen een goed gestructureerd systeem is het mogelijk de betekenis en functie van elke module, onafhankelijk van die van de andere modules te begrijpen. Het lokaliseren van bij elkaar horende functies in één module betekent ook dat een wijziging op slechts een beperkt aantal modules effect heeft. Als de modules tenslotte doelmatig gekozen zijn dan zullen ook de communicaties en verbindingen tussen modules beperkt zijn en ook dat komt de betrouwbaarheid ten goede.

Uniformiteit, volledigheid en testbaarheid

Abstractie en modulariteit zijn belangrijk maar niet voldoende voor het beheersen van de complexiteit van onze software. Deze principes geven immers geen garantie dat het systeem ook correct is. Correctheid en logische consistentie kan worden bevorderd door het toepassen van de principes van uniformiteit, volledigheid en testbaarheid.

Uniformiteit leidt tot begrijpelijkheid. *Uniformiteit* betekent het gebruiken van een eenduidige notatie en analoge structuur van modules [16]. Een goede programmeerstijl, de wijze van toepassen van besturingsstructuren en een eenduidige voorstellingswijze van de gebruikte objecten leiden tot een uniform systeem. In volgende hoofdstukken en in appendix B zullen we met behulp van voorbeelden een dergelijke programmeerstijl illustreren.

Volledigheid en testbaarheid zijn criteria die de betrouwbaarheid, de doelmatigheid en de wijzigbaarheid van ons systeem bevorderen en die correcte oplossingen waarschijnlijker maken. Toepassen van de abstractiemethode helpt ons de essentiële componenten te elimineren; het *volledigheidsprincipe* moet ons ervan verzekeren dat alle benodigde componenten ook aanwezig zijn. Abstractie en volledigheid tesamen helpen ons met het vaststellen van de nodige en voldoende modules op elk niveau. Dit bevordert ook de efficiëntie, omdat nu modules op een lager niveau verbeterd en versneld kunnen worden, zonder dat veranderingen op een hoger niveau hoeven te worden aangebracht.

Het systeem moet verder zo zijn opgebouwd dat het gemakkelijk te testen is [17]. Deze principes zijn niet gemakkelijk toe te passen. Programmeertalen waarin datatypen expliciet moeten worden gedefinieerd kunnen de *testbaarheid* bevorderen, maar we zullen nog zien dat ook software managementgereedschap nodig is om onze systemen volledig en testbaar te maken.

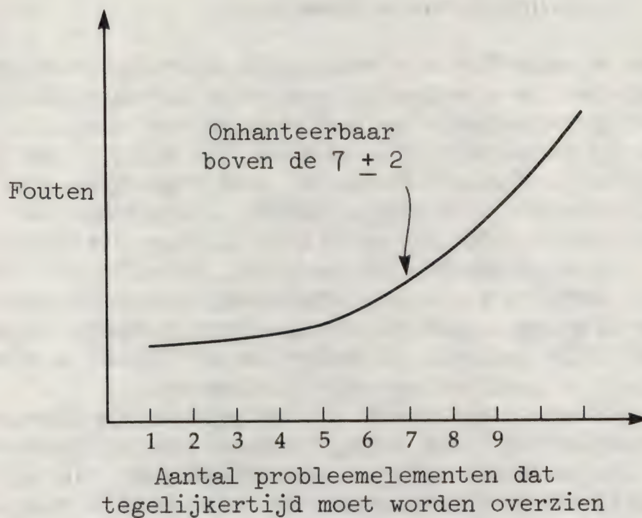
4.3 Software-Ontwikkelingstechnieken





In hoofdstuk 2 gaven we al aan dat het voor de mens moeilijk is een groot aantal verschillende objecten of begrippen tegelijkertijd te overzien. In 1954 stelde de psycholoog George Miller vast dat wij mensen hoogstens tussen de vijf en negen probleemelementen parallel kunnen verwerken [18]. In figuur 4-1 is aangegeven hoe het aantal fouten sterk toeneemt als meer dan dit aantal elementen tegelijk overzien moet worden [19]. Deze complexiteitslimiet wordt wel de *Hrair limiet* [20] genoemd (zie ook de woordenlijst).

Het ontwikkelen van softwaresystemen betekent problemen oplossen en de Hrair limiet lijkt hierop eenduidig van toepassing. De behandelde basisprincipes voor software-ontwikkeling kunnen ons nu helpen systemen zo op te delen in deelsystemen, dat het aantal



Figuur 4-1 Foutencurve bij probleemoplossing (Uit Edward Yourdon/ Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1979, p. 69).

te behandelen grootheden steeds onder de Hrair limiet blijft. Hier zullen we een aantal ontwerpmethodes en beheerstechnieken nader bestuderen, die op deze principes zijn gebaseerd.

Ontwerpmethodes

De software engineering principes kunnen niet zomaar lukraak worden toegepast. We moeten onze systemen op een gedisciplineerde manier structureren. Hierbij is vooral belangrijk dat we logische en elkaar niet tegensprekende criteria gebruiken bij het onderverdelen van ons systeem in modules.

Er zijn vier ontwerpmethodes die dergelijke criteria verschaffen:

- Top-down gestructureerd ontwerp
- Datastructuur ontwerp
- Het Parnas decompositiecriterium
- Object-georiënteerd ontwerp.

Volledige behandeling van de eerste drie methodes valt buiten het kader van dit boek, maar in de volgende paragrafen zullen we de hoofdzaken aangeven.

Top-down gestructureerd ontwerp, zoals beschreven door E. Yourdon, behelst een decompositie van het systeem, zodanig dat elke uit te voeren stap tot een module wordt [21]. Dit leidt tot buitengewoon functionele modules en de methode is zeer geschikt voor sequentiële (stapsgewijze volgtijdelijke) problemen. Op het hoogste abstractieniveau wordt gedefinieerd 'wat' het systeem moet doen. De lagere niveaus bevatten de elementaire operaties, die het 'hoe' mogelijk maken.

D. Jackson en P. Warnier ontwikkelden een andere ontwerpmethode, *data-structuurontwerp* genaamd, die vooral bij COBOL-toepassingen effectief is gebleken [22,23]. Bij deze techniek worden eerst de datastructuren gedefinieerd, en vervolgens worden de programmacomponenten ontwikkeld, gebaseerd op deze datastructuren. Op deze wijze tracht men eerst de objecten vast te stellen die bij de oplossing een rol spelen en vervolgens de functionele eenheden die de operaties bevatten die op die objecten moeten worden uitgevoerd.

Bij het *Parnas decompositiecriterium* wordt het systeem zo onderverdeeld dat elke module een ontwerpbeslissing dekt [24]. Op deze manier is de structuur van het ontwerp terug te vinden op het niveau waar de ontwerpbeslissingen werden genomen. De effecten van wijzigingen zijn dan gemakkelijker te overzien en te lokaliseren.

De vierde ontwerpmethodiek: *object-georiënteerd ontwerp* wordt in het volgende hoofdstuk besproken. In het kort komt de methode er op neer dat eerst de abstracte objecten op het niveau van de oplossingsformulering worden geïdentificeerd, vervolgens de bijbehorende operaties en tenslotte de module, die het oplossingsmechanisme bevat. Deze methode kwam voort uit het werk van een aantal mensen, zoals D. Parnas van de universiteit van North Carolina, B. Liskov en J. Guttag van M.I.T. en Robinson en Leavitt van het Stanford Research Institute. De methode is ook geïnspireerd op de objectmatige benadering, zoals die mogelijk is in talen als SIMULA 67 en SMALLTALK. De specifieke techniek die we hier zullen behandelen voor het ontwerpen van objectgerichte software berust in belangrijke mate op het werk van R. Abbott van de California State University, Northridge.

De management problematiek

Welke methode we uiteindelijk toepassen, de menselijke beperkingen ten aanzien van het overzien van complexe situaties (de Hrair limiet) blijven gelden. Omdat het automatisch genereren van programma's (nog?) niet mogelijk is, althans niet voor praktijkproblemen, moeten onze ontwerpmethodes worden ondersteund door diverse beheerstechnieken. Volledige behandeling van dit onderwerp valt buiten het kader van dit boek en we geven daarom slechts een aantal literatuurverwijzingen.

Het beheersen van een software-ontwikkeling verschilt in problematiek niet van het management van andere grote bouwprojecten,

hoewel het produkt ongetwijfeld minder tastbaar is dan bijvoorbeeld een brug of een schip. In veel gevallen gaat het bij software ook om eenmalige en vaak ook veel ingewikkelder problemen. In hoofdstuk 23 gaan we in op een levenscyclusbenadering van Ada software, maar we kunnen hier alvast stellen dat er voor iedere fase van de levenscyclus een aantal managementgereedschappen beschikbaar zijn.

Tijdens de probleemanalyse en bij het formuleren van het ontwerp kan men bijvoorbeeld gebruik maken van SADT (*Structured Analysis and Design Technique*) [25]. De systeemanalyse en ontwerpbeschrijving kan ook tot stand komen met gebruik van *gegevensstroomdiagrammen* [26] of *structuurschema's* [27]. Ter nadere definitie van de functies in ons ontwerp kunnen we een *procesontwerptaal* (pseudocode) gebruiken. *Gestructureerde inspectie* (structured walkthrough) tenslotte, is een krachtig gereedschap voor het management, [28]. Hierbij wordt het werk van de programmeur aan een intensief en (opbouwend) kritisch onderzoek onderworpen, teneinde modules op volledigheid en testbaarheid te controleren.

4.4 Gereedschappen Voor Software-Ontwikkeling





Om computeroplossingen te creëren is nog wat meer nodig dan alleen een methode. We moeten beschikken over geschikt gereedschap, met name in de vorm van programmeertalen, om ons ontwerp te beschrijven en uitvoerbaar te maken. Peter Wegner heeft in onderstaande lijst een aantal van de meest gebruikte programmeertalen in generaties onderverdeeld en er een aantal van hun specifieke kenmerken aan toegevoegd [29]:

- *Eerste Generatie Talen* (1954-1958)
  FORTRAN I
  ALGOL 58
  Flowmatic
  IPL V

- *Tweede Generatie Talen* (1959-1961)

FORTRAN II	subroutines, afzonderlijke compilatie
ALGOL 60	blokstructuur, datatypen
COBOL	databeschrijving, bestandsorganisatie
LISP	lijstverwerking, pointers

- *Derde Generatie Talen* (1962-1970)

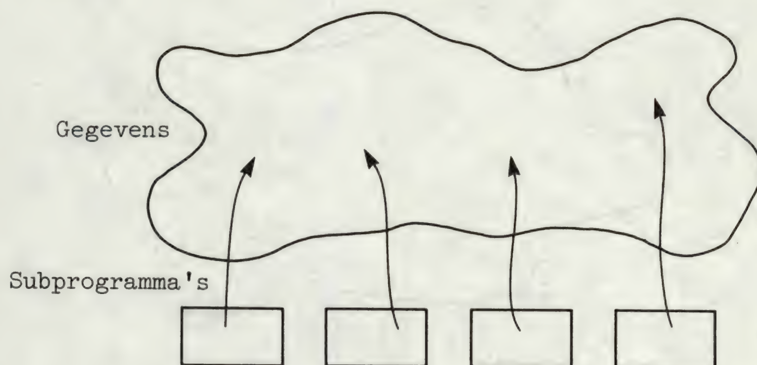
PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	sterk geformaliseerde opvolger van ALGOL 60
Pascal	vereenvoudigde en verbeterde opvolger van ALGOL 60
SIMULA	het klasseconcept en abstracte gegevensvoorstelling

- *De Generatiekloof* (1970-1980)
  Veel verschillende talen, maar geen enkel 'blijvertje'.

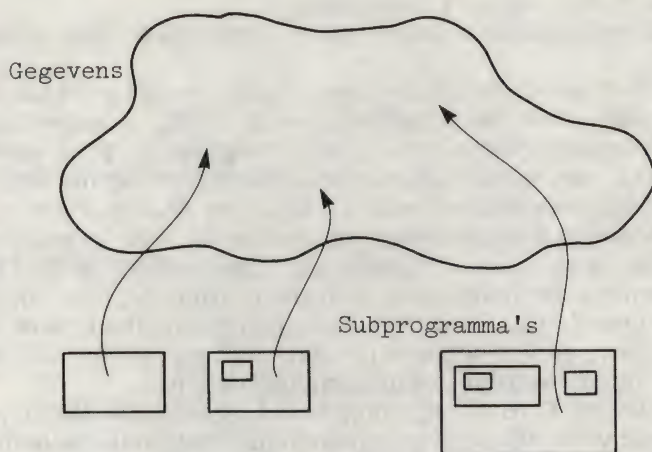
Uit de lijst blijkt dat de nu nog steeds meest gebruikte programmeertalen, varianten van FORTRAN en COBOL, stammen uit de oertijd van de computerwetenschap, lang voordat de problematiek van het ontwerpen van grote softwaresystemen onderkend werd. Deze talen weerspiegelen daarom niet de moderne ideeën over ontwerpmethodes en werden uitgebouwd en misvormd met 'preprocessors' (zoals S-FORTRAN), en uitbreidingen (zoals FORTRAN-77), om ze bij de modernere methodes aan te passen. Hoe dit ook zij, zeker is dat de problemen, waarmee men zich bezighield toen deze talen ontworpen werden, een grootte-orde eenvoudiger waren dan de problemen uit de tegenwoordige toepassingsgebieden.

FORTRAN en COBOL zijn nog steeds geschikte talen voor de gebieden waarvoor zij werden ontworpen; inmiddels is echter het volledig nieuwe gebied van de ingebouwde ('embedded') computersystemen ontstaan. Noch FORTRAN, noch COBOL, noch de meeste andere hedendaagse programmeertalen zijn voor dit gebied geschikt en zij zijn ook niet berekend op het ontwerpen van software die bestaat uit duizenden of tienduizenden programmaregels.

Als we de topologie van die eerste generatie talen bestuderen dan wordt duidelijk waarom er problemen ontstaan. Zoals in figuur 4-2 te zien is, hebben zowel FORTRAN als COBOL een betrekkelijk vlakke structuur, die bestaat uit globale gegevens en één niveau van subprogramma's. Een fout op één plaats in een programma kan een verwoestende kettingreactie tot gevolg hebben door de rest van het systeem vanwege de globale datastructuur. Bij wijzigingen is het verder vrijwel onmogelijk om de structuur van het oorspronkelijke ontwerp te behouden. Zelfs na een gering aantal wijzigingen tijdens de onderhoudsperiode ontstaan er ongewenste kruisverbanden tussen programma-eenheden en zo vermindert de betrouwbaarheid en neemt in ieder geval de helderheid van de oorspronkelijke oplossing sterk af.



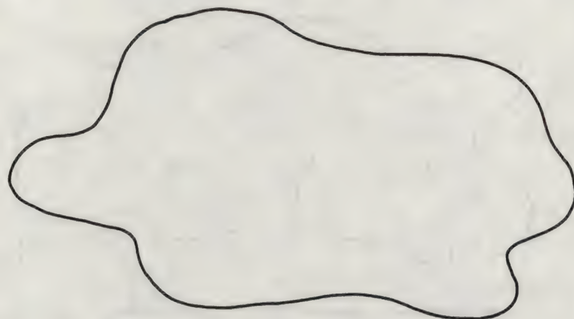
Figuur 4-2 Topologie van eerste en tweede generatie talen



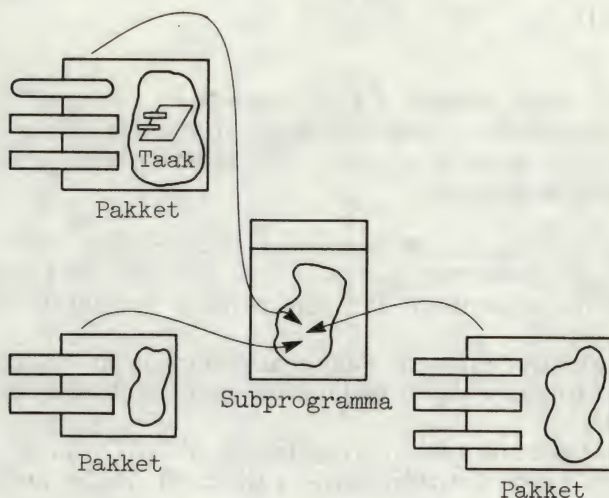
Figuur 4-3 Topologie van tweede en derde generatie talen

De ontwikkeling van ALGOL 60 in de tweede generatie maakte het mogelijk algoritmen te formuleren met behulp van overzichtelijker 'geneste' structuren, maar ten aanzien van structurering van data waren er nog weinig mogelijkheden. Dit laatste geldt ook nog voor de meeste derde generatie talen. In figuur 4-3 is aangegeven hoe hun topologie maar betrekkelijk weinig verschilt van de eerste generatie. Bij enkele talen bestaan meer mogelijkheden voor het definiëren van gegevensstructuren, we noemen SIMULA, Alphard, CLU en LIS, maar geen van die talen werd in brede kring geaccepteerd.

In figuur 4-4 hebben we de topologie van assembleertalen weergegeven; deze talen worden nu nog steeds veel gebruikt voor het bouwen van 'embedded' systemen. Het plaatje lijkt misschien vrij absurd, maar is bedoeld om aan te geven dat assembleertalen geen



Figuur 4-4 Topologie van assembleertalen



Figuur 4-5 De topologie van Ada

enkele structuur hebben. Weliswaar is hun flexibiliteit zeer groot en is het best mogelijk er gestructureerd in te programmeren, maar zelfs bij een matig groot probleem wordt de oplossing bij het gebruik van een assembleertaal al zeer gecompliceerd.

Ada werd ontwikkeld tegen het einde van de programmeertaal 'generatiekloof' en is daarom sterk beïnvloed door de huidige software-ontwikkelingsmethodes. Het is in bepaalde opzichten de eerste vierde generatie taal. Figuur 4-5 laat zien dat Ada's topologie niet vlak is, zoals bij eerdere generaties, maar multidimensionaal. We kunnen nu onze algoritmen en onze gegevens volgens vaste regels structureren. Het is nu ook mogelijk de complexiteit van onze oplossingen te beheersen door het fysiek ontoegankelijk maken van de details van lagere abstractieniveaus. Omdat ontwerpbeslissingen op een bepaald niveau gelokaliseerd kunnen worden kan de structuur van het ontwerp ook bij latere wijzigingen gemakkelijk behouden blijven.

In de volgende hoofdstukken zullen we nader ingaan op de kenmerken van Ada, die een directe ondersteuning vormen voor de basisprincipes van software engineering, zodat het mogelijk wordt oplossingen te formuleren die wijzigbaar, efficiënt, betrouwbaar en begrijpelijk zijn. In hoofdstuk 5 introduceren we een object-georiënteerde ontwerpmethode, waarin van die principes gebruik wordt gemaakt en waarin de kracht van Ada's topologie duidelijk naar voren komt.

Oefeningen

1. Bekijk de kamer waarin u zich bevindt en beschrijf een object volgens de methode van afnemende abstractie. Kies bijvoorbeeld een stoel, een bureau of een telefoon en onderscheid minstens tien abstractieniveaus.

*2. Beschouw de vier in dit hoofdstuk beschreven ontwerpmethodes. Welke programmeertaal past het best bij elke methode? Zijn er ook talen die voor meer dan één methode geschikt zijn?

3. Kies een softwareproject waarin u deelnam. Welke managementgereedschappen werden tijdens de ontwikkeling gebruikt?

4. Neem een bekende taal, bijvoorbeeld JOVIAL, C, of zelfs BASIC, en geef de topologie schematisch weer. In welke generatie zou u de taal plaatsen?

5 OBJECT-GERICHT ONTWERP

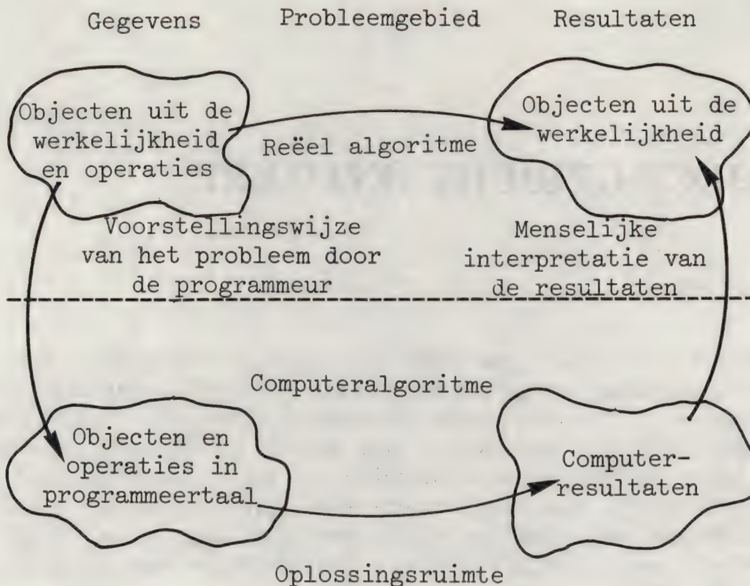
In het vorige hoofdstuk formuleerden wij de doelstellingen van software engineering en gaven we enkele eenvoudige principes aan die ons kunnen helpen die doelstellingen te bereiken. Het verhaal bevatte één duidelijke boodschap: we moeten geschikte gereedschappen gebruiken en geschikte technieken om de complexiteit van grote softwaresystemen te kunnen beheersen. In dit hoofdstuk wordt zo'n ontwerptechniek behandeld en zullen we laten zien hoe de methode gebruik maakt van en ondersteund wordt door de specifieke eigenschappen van Ada.

5.1 De Beperkingen Van De Functionele Aanpak



Welke toepassing we ook bekijken, het probleemgebied heeft altijd zijn oorsprong in de werkelijkheid en de oplossingsruimte wordt gevormd door een combinatie van programma's en apparatuur.
H. Ledgard ontwikkelde een model om de taak van de programmeur te beschrijven en dit wordt schematisch weergegeven in figuur 5-1 [1]. In het probleemgebied bevindt zich een aantal objecten uit de werkelijkheid, waarmee een aantal bijbehorende bewerkingen verbonden is. Het kan daarbij gaan om iets eenvoudigs als een kasboek of iets heel ingewikkelds zoals een ruimtevaartuig. In het probleemgebied bevinden zich ook concrete algoritmen, die op deze objecten inwerken en als resultaten getransformeerde objecten opleveren. Zo'n resultaat kan bijvoorbeeld zijn de balans van een kasboek of een koersaanpassing van een ruimtevaartuig.

Als we een softwaresysteem ontwikkelen dan maken we ofwel een volledig model van ons probleem uit de realiteit, ofwel in het geval van in een groter systeem ingebouwde computersystemen is er sprake van een combinatie van reële objecten en software, die tesamen tot reële resultaten leiden. Hoe dan ook, steeds is er sprake van een analogie tussen probleemgebied en oplossingsruimte. De eerste stap is dat een programmeertaal de programmeur het gereedschap verschaft om objecten uit de werkelijkheid voor te stellen; de programmeur bedenkt een abstracte voorstellingswijze voor de objecten en



Figuur 5-1 Model voor de taak van de programmeur. (Uit *The Programming Language Landscape* door Henri Ledgard en Michael Marcotty, 1981, Science Research Associates, Inc.)

formuleert deze abstractie in zijn programma. Vervolgens wordt een algoritme toegepast dat deze programma-objecten transformeert. Opnieuw gebruikt de programmeur daarbij haar of zijn abstracte voorstelling van bewerkingen uit de werkelijkheid. Uiteindelijk worden resultaten geproduceerd, die ofwel direct worden afgebeeld op een actie in de werkelijkheid, ofwel door mensen achteraf worden geïnterpreteerd.

Intuïtief is het duidelijk dat, hoe dichter de oplossingsruimte staat bij onze voorstelling, des te beter kunnen de doelstellingen als wijzigbaarheid, efficiëntie, betrouwbaarheid en begrijpelijkheid worden bereikt. Zoals in hoofdstuk 4 werd gesteld: van alles dat we waarnemen maken we abstracties en hoe verder de oplossingsruimte afstaat van het probleemgebied, des te groter is de stap naar onze voorstelling van de werkelijkheid en dat doet de complexiteit van onze oplossing toenemen.

Alle natuurlijke talen kennen zelfstandige naamwoorden en werkwoorden. In programmeertalen bestaat iets overeenkomstigs, want er kunnen objecten (zelfstandige naamwoorden) en bewerkingen (werkwoorden) in worden beschreven. De meeste programmeertalen die aan Ada vooraf gingen kenden echter alleen de gebiedende wijs: zij beschikken over legio mogelijkheden om bewerkingen of operaties te formuleren, maar hebben weinig middelen om abstracties van objecten voor te stellen. Al die talen hebben topologisch gezien een betrekke-

lijk vlakke structuur, terwijl de werkelijkheid niet vlak is en niet volgtijdelijk of sequentieel, maar integendeel multidimensionaal en meestal in hoge mate gelijktijdig of parallel. De eerste drie generaties programmeertalen, en zeker assembleertalen, vergrootten juist de kloof tussen probleemgebied en oplossingsruimte.

In het vorige hoofdstuk gaven we in het kort de principes weer van drie gebruikelijke software-ontwerpmethoden:

- top-down ontwerp
- datastructuurontwerp
- Parnas decompositiecriterium

Top-down ontwerpmethoden zijn van nature gebiedend of imperatief: zij dwingen ons om ons te concentreren op de bewerkingen in de oplossingsruimte en bekommeren zich weinig om de datastructuren. De datastructuur-ontwerptechnieken gaan juist uit van het andere uiterste: zij concentreren zich op de objecten en behandelen de benodigde bewerkingen juist globaal. Het Parnas decompositiecriterium maakt het wel mogelijk zowel objecten als operaties te behandelen, maar leidt wegens de beperktheid van de meeste programmeertalen toch meestal tot een oplossing naar functies.

Als we dit soort ontwerpmethodes gebruiken is het resultaat ofwel een volledig functiegerichte oplossing, waarin het model van de werkelijkheid niet meer terug is te vinden, ofwel een oplossing met duidelijke gegevensstructuren, maar met onduidelijk geformuleerde bewerkingen. Dit valt te vergelijken met een gesprek voeren terwijl je ofwel alleen werkwoorden, ofwel alleen zelfstandige naamwoorden gebruikt. Op zijn minst is een gedachtensprong nodig om de oplossingsruimte terug te vertalen naar het probleemgebied en in het ergste geval moet ook een fysieke transformatie worden uitgevoerd. Al deze methodes tenslotte, dwingen ons tot sequentieel denken en bieden geen middelen om gelijktijdige gebeurtenissen of processen uit het probleemgebied voor te stellen.

5.2 Een Object-Gerichte Ontwerpmethode





Wat we willen is een ontwerpmethode die een directe afbeelding van het probleemgebied naar de oplossingsruimte mogelijk maakt. Net zoals in natuurlijke talen moet daarbij een combinatie van objecten en bewerkingen mogelijk zijn. We zullen dit een *object-gerichte* ontwerpmethode noemen om nadruk te leggen op het feit dat het hier geen zuivere functiegerichte methode betreft. Deze benadering erkent het belang van programma-objecten, elk met zijn eigen verzameling toegelaten operaties.

De stappen waaruit onze methode bestaat kunnen als volgt worden samengevat:

- *Definieer het probleem.*
- *Ontwikkel een informele oplossingstrategie.*
- *Formaliseer de strategie*
  Identificeer de objecten en hun kenmerken
  Identificeer de bewerkingen op de objecten
  Stel de communicatiepatronen vast
  Programmeer de bewerkingen.

Het was Abbott (zie ook hoofdstuk 7), die als eerste deze stappen formuleerde en het verband legde tussen zelfstandige naamwoorden en objecten en tussen werkwoorden en operaties in zijn Ada Stijl Gids [3]. Onze bijdrage is een voortzetting van Abbotts werk, speciaal gericht op Ada en in het bijzonder op complexe systemen met parallelle processen.

Definieer het probleem

We beginnen onze methode zoals gebruikelijk met de formulering van de probleemstelling. Tot hier is er geen verschil met traditionele methodes, en hulpmiddelen zoals SADT of gegevensstroomdiagrammen kunnen dan ook tot aan dit punt gebruikt worden. Belangrijk is dat we inzicht verkrijgen in de structuur van het probleemgebied.

Vanzelfsprekend kan men niet vanaf het begin een volledig inzicht hebben in het probleem. Er is hier sprake van een iteratief proces en naarmate we dieper ingaan op het ontwerp van de oplossingsmethode, zullen we hoogstwaarschijnlijk ook nieuwe en nog onvoorziene aspecten van het probleem ontdekken. Omdat we er echter naar streven een oplossing te ontwikkelen met een directe afbeelding naar het probleemgebied, zullen deze nieuw ontdekte aspecten niet van grote invloed zijn op het ontwerp. Hoogstens moet een module worden aangepast die betrekking heeft op een grootheid uit het probleemgebied. Meestal wordt het probleem beschreven met behulp van een formeel specificatiesysteem; in onze voorbeelden zullen we alleen die probleemaspecten introduceren, die nodig zijn om de oplossing op een bepaald abstractieniveau volledig te formuleren.

Ontwikkel een informele oplossingstrategie

Zodra we inzicht hebben in de hoogste abstractieniveaus van het probleem, kunnen we beginnen met het formuleren van een informeel aanvalsplan. Hoewel men tegenwoordig alles wil formaliseren is de menselijke manier van denken nu eenmaal niet formeel. We hebben meestal eerst een intuïtief idee over het probleem en proberen dat

dan steeds beter te begrijpen door de beschrijving meer en meer te formaliseren. In dit verband merken R. Balzer, N. Goldman en D. Wile op, dat "... erkend moet worden dat er altijd een informeel stadium zal zijn tijdens het formuleren van een specificatie" [2].

Uiteindelijk is een formele formulering nodig als we het probleem volledig en zonder tegenstrijdigheden willen beschrijven aan een mens of aan een computer. Onze benadering zal een dergelijke formele behandeling dan ook mogelijk maken. Vooralsnog zullen we echter gebruik maken van ons intuïtieve inzicht in problemen. Met behulp van de object-gerichte ontwerpmethode zullen we onze informele oplossingsstrategie formuleren in gewoon Nederlands en in termen van de probleemstelling. In dit stadium moet de ontwerper zich bij voorkeur nog geen beperkingen opleggen wat betreft de vorm van de beschrijving. Omdat het gaat om een beschrijving van de werkelijkheid, is dit niet het juiste moment om onszelf beperkingen op te leggen in het nadenken over het probleem. Ook de structuur van de oplossing komt hier nog niet aan bod; de ontwikkeling zal op natuurlijke wijze plaatsvinden.

Formaliseer de strategie

De derde stap is het formaliseren van onze oplossingsstrategie met behulp van enkele eenvoudige regels. Eerst identificeren we de objecten en hun kenmerken. We vermeldden al dat programmeertalen constructies kunnen bevatten waarmee objecten kunnen worden beschreven, zoals zelfstandige naamwoorden in natuurlijke talen. Om deze stap uit te voeren keren we terug naar onze informele formulering en zoeken alle zelfstandige naamwoorden op die objecten voorstellen en alle daaraan toegevoegde bijvoeglijke naamwoorden, die de kenmerken van die objecten aangeven. Dit is een betrekkelijk mechanisch proces, dat misschien wel geautomatiseerd zou kunnen worden met behulp van een kunstmatige intelligentiemethode.

Er zijn een aantal typen van zelfstandige naamwoorden te onderscheiden [3]:

- *Algemene zelfstandige naamwoorden*. Namen van klassen van grootheden (bijvoorbeeld: tafel, beeldbuis, schakelaar).
- *Zelfstandige naamwoorden met een hoeveelheidsaspect*. Namen van stoffen en substanties (water, brandstof).
- *Echte zelfstandige naamwoorden*. Namen van specifieke grootheden (drukmeter, reset-schakelaar).

De eerste twee categorieën hebben geen betrekking op specifieke objecten, maar op klassen van objecten of abstracte datatypen (zie hoofdstuk 8). Specifieke objecten kunnen worden aangeduid als verschijningsvormen uit die klassen. De derde categorie heeft direct betrekking op specifieke objecten uit ons probleemgebied.

Het selecteren van bijvoeglijke naamwoorden bij de objecten betekent het toekennen van eigenschappen of kwalificaties. Het kan hier gaan om beperkingen of randvoorwaarden (zoals het interval van mogelijke waarden), terwijl er ook tijdsafhankelijkheden door kunnen worden uitgedrukt. Als we bijvoorbeeld kwalificaties zoals 'asynchroon', 'gelijktijdig' of 'onderling onafhankelijk' gebruiken, dan is het zelfs in dit vroegtijdige stadium mogelijk om aan te geven dat het probleem aspecten van gelijktijdigheid of parallelliteit bevat.

De volgende stap is een herhaling van onze mechanische procedure, maar nu ter identificatie van de werkwoorden in onze informele strategiebeschrijving. Zo is het mogelijk een overzicht te krijgen van de bewerkingen en elke bewerking moet daarbij geassocieerd worden met een bepaald object. Zo worden kenmerken van de operaties, zoals volgorde van uitvoering, tijdsaspecten en aantal iteraties vastgesteld. Hier kan al blijken dat er sprake is van processen die parallel verlopen met andere processen.

Onze strategie kan nu nog verder worden geformaliseerd. Nu kunnen de relaties tussen objecten worden vastgesteld. Dat betekent dat we nu de raakpunten, communicatiepatronen of interfaces tussen objecten formeel kunnen beschrijven. Hierbij kan Ada al als ontwerptaal worden gebruikt. We sluiten zo als het ware een contract tussen de gebruiker van een object en het object zelf, waarbij de toegelaten bewerkingen expliciet worden beschreven. In de vorige stap werd immers al vastgesteld welke bewerkingen een rol spelen. In het vervolg zullen we nog zien, dat Ada het niet alleen mogelijk maakt een dergelijk contract te formuleren, maar dat Ada ons ook dwingt het contract na te leven, door het fysiek onmogelijk te maken de geformuleerde logische abstracties willekeurig te doorbreken. Een bepaald object in de realiteit staat niet in directe relatie tot alle andere objecten, dat betekent ook dat de reikwijdte ('scope') en bereikbaarheid van elke grootheid formeel moet worden beschreven. De hiertoe benodigde gegevens kunnen direct worden bepaald op grond van onze informele strategie. In het volgende hoofdstuk introduceren we grafische hulpmiddelen om reikwijdte en bereikbaarheid van grootheden weer te geven.

De volgende stap is het formuleren van de bewerkingen op de objecten met behulp van onze programmeertaal. Op deze manier wordt een ontwerp geformuleerd dat meteen ook uitvoerbaar is. De informele strategiebeschrijving geeft daarbij aan in welke volgorde de bewerkingen moeten worden uitgevoerd. Beschrijft de informele strategie daarbij gelijktijdig plaatsvindende gebeurtenissen, dan kunnen deze nu in de vorm van parallel uitgevoerde programmataken worden beschreven.

Hiermee is ons proces nog niet voltooid, want tijdens het uitvoeren van de stappen zullen we zeker nieuwe objecten en operaties ontdekken, die nog moeten worden toegevoegd. Dit betekent dat we het proces op een nieuw abstractieniveau moeten herhalen en opnieuw objecten en bewerkingen moeten identificeren.

We zullen nog geen voorbeeld van de methode geven, maar eerst de benodigde gereedschappen bestuderen, zoals die in onze

programmeertaal zijn opgenomen. Toch begint nu al duidelijk te worden hoe onze benadering de principes van goede software-ontwikkeling kan ondersteunen. De object-gerichte ontwerpmethode maakt gegevensabstractie en beperkt toegankelijk maken van gegevens mogelijk. Daarbij kunnen we met behulp van Ada de onderliggende details van onze bewerkingen inderdaad fysiek ontoegankelijk maken en datzelfde is mogelijk met de fysieke voorstellingswijze van onze objecten.

De benadering biedt een doelgerichte strategie voor het opdelen van een systeem in modules, waarbij onze ontwerpbeslissingen zo gelokaliseerd worden dat sprake blijft van een overeenkomst met de waargenomen werkelijkheid. We zullen wel nog steeds gebruik moeten blijven maken van management gereedschappen om het ontwikkelingsproces te beheersen en de volledigheid en testbaarheid te waarborgen, maar onze taak wordt nu wellicht iets gemakkelijker, omdat via de object-georiënteerde benadering een bepaalde structuur als het ware opgelegd kan worden.

5.3 Van Ontwerp Naar Ada



Elke taal, natuurlijke taal of programmeertaal doet twee dingen. Ten eerste biedt een taal een *bereik van uitdrukkingsmogelijkheden*. Een bepaalde Eskimotaal heeft bijvoorbeeld 30 woorden voor 'sneeuw'. Evenzo is het in een taal als APL mogelijk te denken in termen van vectoren. Ten tweede legt de taal *beperkingen* op aan de manier van denken van de gebruiker. Engels bevat bijvoorbeeld zeer veel mogelijkheden om acties te beschrijven; veel andere Europese talen bevatten juist meer zelfstandige naamwoorden. Vraag ook bijvoorbeeld eens aan een FORTRAN programmeur om een probleem met behulp van recursie op te lossen.

Elke programmeertaal zou voldoende hulpmiddelen moeten bieden om oplossingen te kunnen formuleren. De taal zou zodanige uitdrukkingsmiddelen moeten hebben dat het probleemgebied direct kan worden weergegeven. Bij de eerdere generatie programmeertalen moest vaak het probleem aan de taal worden aangepast, in plaats dat aanpassing van de taal aan de oplossing mogelijk was. Zo staat ons gereedschap in de weg bij het bereiken van ons uiteindelijke doel: oplossen van het probleem. Als daarentegen de programmeertaal zodanig is dat de oplossingsformulering een directe weerspiegeling is van het gestelde probleem, dan is de programmatuur begrijpelijk en helpt ons bij het beheersen van de complexiteit van grotere problemen. Een dergelijke taal moet de mogelijkheid bieden elementaire objecten en bewerkingen te beschrijven en moet zodanig uitbreidbaar zijn, dat wij onze eigen abstracte objecten en operaties daarop kunnen formuleren. Idealiter dwingt de taal ons ertoe, van abstracties gebruik te maken en maakt het onmogelijk ze te doorbreken.

Zo'n taal is nu Ada. Ada beschikt over een uitgebreide verzameling van constructies, waarmee elementaire objecten en operaties kunnen worden beschreven en biedt de mogelijkheid van het creëren van pakketten, zodat we onze eigen abstracte objecten kunnen construeren en gebruiken. In het volgende hoofdstuk zullen we de technische aspecten van de taal uitgebreid gaan behandelen, met illustraties door middel van onze object-gerichte ontwerpmethode. Natuurlijk is deze ontwerpmethode niet de enige die in combinatie met Ada kan worden gebruikt; elke conventionele ontwerpmethode zou kunnen worden toegepast. Maar Ada heeft voor de programmeur enkele unieke eigenschappen die je tot nu toe niet vindt in de meeste andere programmeertalen: het definiëren van taken, het correct ondervangen van uitzonderingen en het pakketprincipe. Ada vereist daarom dat wij loskomen van de gebruikelijke vlakke sequentiële denkwijze, zoals die ons door andere programmeertalen wordt opgelegd, en dat we leren denken in termen van de probleemstelling. De object-gerichte ontwerpmethode levert daartoe het conceptuele model.

Oefeningen

1. Stelt u zich een appel voor als een abstract datatype. Welke bewerkingen zijn er van toepassing op objecten van het type appel? Beschouw verschillende gebruikers van de appel: de boer, de groenteman en de consument.

2. Kunnen abstracties ook met behulp van een assembleertaal worden gecreëerd? Licht uw antwoord toe.

*3. Schaf u een handboek of definitie aan van de talen SMALLTALK, CLU en LIS, of van Pascal. Op welke wijze kunnen in deze talen gegevensabstracties en functionele abstracties worden gecreëerd?

6 ADA, EEN OVERZICHT

Nu we de grondslagen van moderne software-ontwikkelingsmethodes hebben behandeld, kunnen we de taal Ada zelf gaan bestuderen. Deze voorzichtige inleiding zal u een eerste idee geven van de mogelijkheden van Ada en u zult een deel van de structuur en de terminologie van de taal leren kennen. Probeer niet meteen alles te doorgronden: de programmeervoorbeelden zijn slechts bedoeld om de uiterlijke verschijningsvorm van Ada te leren kennen. Verwacht niet alle details van de voorbeelden volledig te begrijpen. Een uitgebreide behandeling van alle constructies volgt later. We zullen om te beginnen het principe van beperkte toegankelijkheid van gegevens toepassen en slechts het hoogste abstractieniveau van de taal bestuderen.

6.1 Een Eisenpakket Voor De Taal





Grif de volgende woorden in uw geheugen: *Ada is een ontwerptaal.* Dit betekent dat Ada geschikt is om een oplossing te formuleren voor de gehele levenscyclus van een softwareproject (zie ook hoofdstuk 23). Als we het gebied bekijken waarvoor Ada werd ontworpen, is het niet verwonderlijk dat Ada een taal moest zijn met een grote uitdrukkingskracht. Laten we de STEELMAN specificaties voor de taal eens bekijken [1]:

- gestructureerde constructies
- expliciet omschreven datatypes
- relatieve en absolute precisie moet kunnen worden gespecificeerd
- parallelle verwerking moet mogelijk zijn
- mogelijkheden tot opvangen van uitzonderingen
- generieke definities
- machine-afhankelijke faciliteiten moeten kunnen worden toegevoegd.

Deze eisen brachten de ontwikkeling van programmeertalen op een hoger niveau en waren er de oorzaak van dat een aantal bestaande concepten tesamen werden gebracht in één nieuwe hogere programmeertaal.

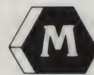
Tijdens het ontwerp van ieder systeem moeten compromissen worden gesloten; Ada was hierop geen uitzondering. Als leidraad voor hun ontwerpbeslissingen koos het Ada-team voor drie doelstellingen [2]:

- erkenning van het belang van betrouwbaarheid en onderhoudbaarheid
- onderkennen van het feit dat programmeren mensenwerk is
- doelmatigheid

Merk op dat betrouwbaarheid en onderhoudbaarheid het eerst werden genoemd. In hoofdstuk 2 zagen we al dat dit de belangrijkste kostenbepalende factoren zijn tijdens de softwarelevenscyclus. Vervolgens wordt programmeren als menselijke activiteit genoemd, met de nadruk op het helpen van de programmeur bij het beheersen van de sofwarecomplexiteit. We zullen zien dat Ada programma's gemakkelijk te lezen zijn, hoewel soms langdradig schrijfwerk nodig is. Toch is dit beter dan andersom: een programma wordt maar één keer geschreven, maar talloze malen gelezen. De regels van de taal brengen de programmeur er in vele opzichten toe een goed ontwerp en een goede programmeerstijl te gebruiken. De Ada programmeeromgeving APSE (Ada Programming Support Environment) vormt een verdere aanvulling op de taal en kan het programmeren nog meer vergemakkelijken. Doelmatigheid van formulering en uitvoering was de derde doelstelling van het ontwerpteam. Constructies met een onduidelijke structuur of die overmatig veel verwerkingstijd vereisten werden verworpen.

6.2 Ada Van Top Tot Teen





Een Ada programma (wij spreken liever van een Ada *systeem*) bestaat uit één of meer programma-*eenheden*, die ieder afzonderlijk kunnen worden gecompileerd. Programma-eenheden bestaan uit subprogramma's, taken en pakketten. Een *subprogramma* is een procedure of een functie en vertegenwoordigt één enkele actie. Een *taak* omschrijft een actie die wordt uitgevoerd tegelijk met andere taken. Taken kunnen worden verwerkt door één enkele processor, een multiprocessor of een computernetwerk. Een *pakket* is een verzameling van rekenmiddelen, en kan datatypen, data-objecten, subprogramma's, taken, of zelfs andere pakketten omvatten. Het voornaamste doel van pakketten is de logische abstracties van de ontwerper te formuleren en hem vervolgens te dwingen er de hand aan de houden. In tabel 6-1

Tabel 6-1: Samenvatting van Ada's programma-eenheden

Programma-eenheid	Karakteristiek	Toepassingen
Subprogramma	Sequentiële verwerking	Hoofdprogramma's Definitie functionele besturing Definitie type operaties
Pakket	Verzameling van hulpmiddelen	Benoemde verzameling declaraties Groepen samenhangende eenheden Abstracte datatypes Abstracte toestands-automaten
Taken	Parallelle actie	Tegelijk verlopende acties Verzenden van berichten Besturen van rand-apparatuur Onderbrekingen (interrupts)

worden de karakteristieken van Ada's programma-eenheden opgesomd tesamen met hun toepassingen.

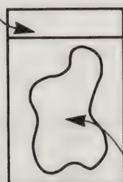
Alle programma-eenheden in Ada hebben dezelfde structuur die bestaat uit twee gedeelten: een specificatiegedeelte en een rompgedeelte. De *specificatie* beschrijft de informatie die toegankelijk is voor de gebruiker van de eenheid (de interface), en de *romp* bevat het mechanisme dat ten opzichte van de gebruiker ontoegankelijk kan worden gemaakt. Om het mogelijk te maken ook grote systemen te ontwikkelen kunnen specificatiegedeelte en romp afzonderlijk worden gecompileerd. Op die manier is het mogelijk eerst de specificaties te schrijven van de programma-eenheden op het hoogste abstractieniveau en pas later de rompgedeelten toe te voegen, die verder kunnen worden uitgewerkt, onafhankelijk van de specificatiegedeelten.

Omdat het bij 'embedded' computersystemen vaak gaat om ingewikkelde problemen, kan een Ada systeem bestaan uit honderden of zelfs duizenden programma-eenheden. Naarmate de omvang toeneemt, wordt het steeds moeilijker een dergelijk systeem te overzien. Met behulp van de notatie die we in hoofdstuk 5 invoerden, kunnen we Ada programma-eenheden met behulp van één enkel symbool weergeven [3].

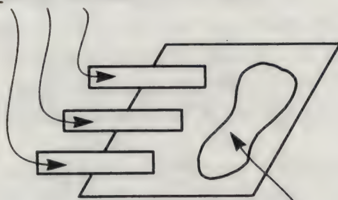
In figuur 6-1 wordt een niet gedefinieerde of volledig ontoegankelijke grootheid (een programma-eenheid of een structuur op een lager niveau) door een vormeloze wolk weergegeven. We geven



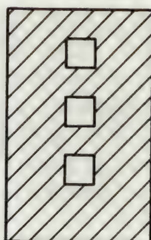
Figuur 6-1 Een niet-gedefinieerde grootheid



Figuur 6-2 Een Ada subprogramma



Figuur 6-3 Een Ada taak

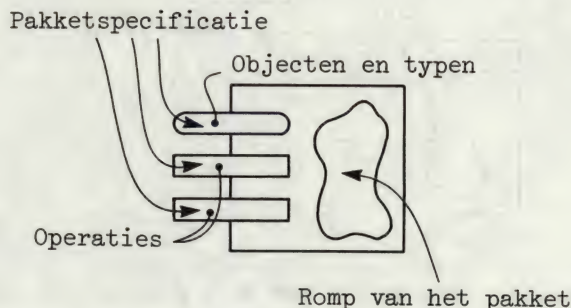


Figuur 6-4 Pakket met toegankelijke onderdelen

hiermee aan dat de structuur voor ons op dit niveau niet relevant is en ontoegankelijk. In figuur 6-2 is een subprogramma voorgesteld door een lineaire structuur, hetgeen zijn volgtijdelijke of sequentiële aard aangeeft. Merk op dat het subprogramma bestaat uit een specificatie en een romp, die weer andere eenheden kan bevatten. Een taak is in figuur 6-3 weergegeven als een parallellogram, een indicatie voor de parallelle of gelijktijdige aard van taken. Ook hier bestaat weer een onderscheid tussen taakspecificatie en de romp van de taak.

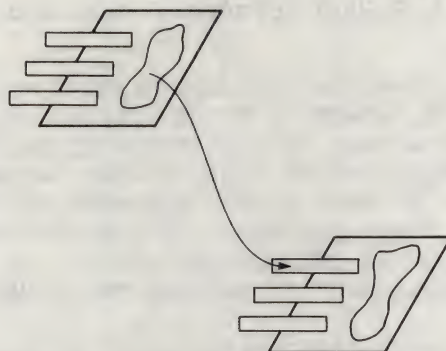
Pakketten zijn heel belangrijke en veelzijdige Ada-structuren en we geven ze aan met een speciaal symbool. Ichbiah, één van de ontwerpers van Ada, beschrijft pakketten als muren, die een aantal logische samenhangende grootheden omringen en afschermen [4]. In figuur 6-4 zijn de toegankelijke gedeelten van het pakket (het specificatiegedeelte) getekend als ramen in de muur. Als we het negentig graden naar links omklappen, dan ontstaat figuur 6-5, een grafische weergave van het buitenaanzicht van een pakket, waarbij we van de ramen alleen de vensterbanken zien.

Ingewikkelde structuren kunnen nu met behulp van deze symbolen worden geïllustreerd. In figuur 6-6 zijn bijvoorbeeld twee onderling communicerende taken getekend. De taken zijn onderling afhankelijk en de pijl wijst van de sturende taak naar de ondergeschikte

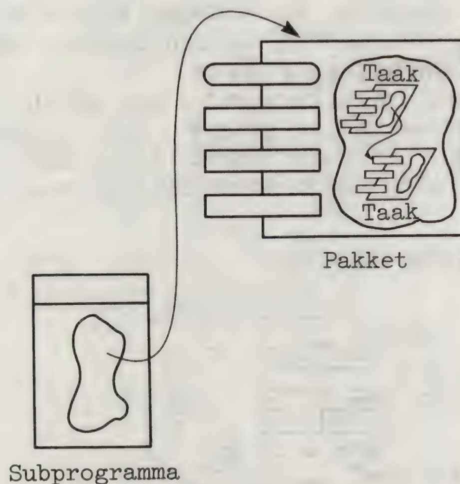


Figuur 6-5 Een Ada pakket

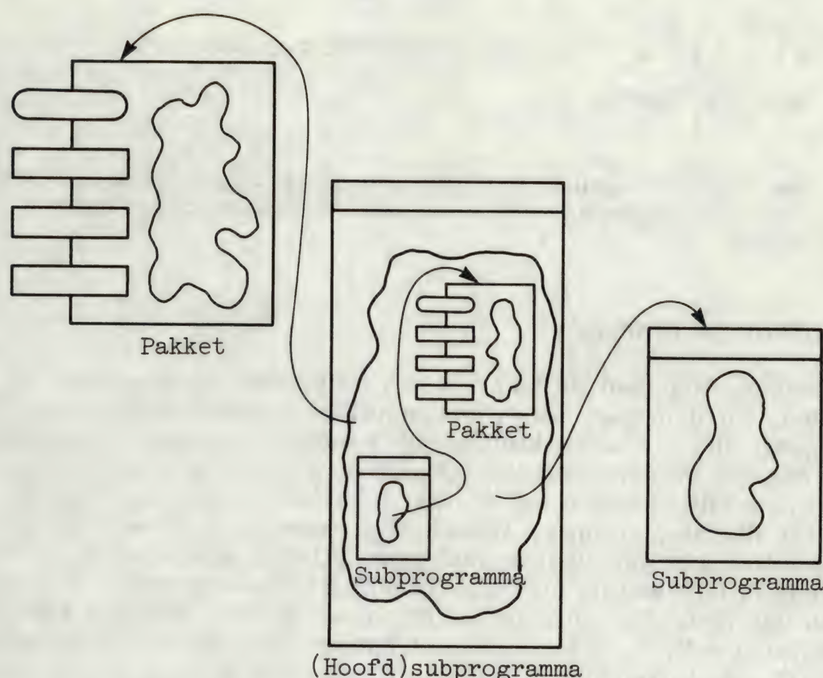
(bedienende) taak. In figuur 6-7 bekijken we het geheel van een abstractieniveau hoger en is iets meer te zien van het totale ontwerp. Hier geeft de pijl aan dat het subprogramma bediend wordt door het pakket. Als we ons naar het hoogste systeemniveau begeven (dit is altijd een subprogramma), dan zijn nog meer lagen te zien. Figuur 6-8 bestaat uit een hoofdprogramma en twee bibliotheekroutines buiten dit hoofdprogramma. Het bibliotheekidee wordt verder behandeld in hoofdstuk 20.



Figuur 6-6 Twee communicerende Ada taken



Figuur 6-7 Het nesten van Ada programma-eenheden



Figuur 6-8 Een Ada programma van bovenaf gezien

6.3 Ada Van Teen Tot Top



Alle Ada constructies worden beschreven met behulp van de volgende basisset van alfanumerieke tekens:

- *Hoofdletters*
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- *Cijfers*
  0 1 2 3 4 5 6 7 8 9
- *Speciale tekens*
  " # ' ( ) * + , - / : ; < = > _ | &
- *De spatie*

STEELMAN specificeerde deze tekenset om overdraagbaarheid van programmatekst mogelijk te maken. Ada maakt het mogelijk de verzameling uit te breiden tot de 95 tekens van de ASCII (American Standard Code for Information Interchange) code:

- *Kleine letters*
  a b c d e f g h i j k l m n o p q r s t u v w x y z
- *Speciale tekens*
  ! $? @ [\ ] ' { } ~ ^ %

Programma's geschreven met behulp van deze uitgebreide tekenset kunnen worden geconverteerd naar een equivalent programma in de basistekenset.

Lexicografische eenheden

De basiselementen van de taal met een betekenis, *lexicografische eenheden*, worden opgebouwd met behulp van Ada's lettertekenset. Lexicografische eenheden kunnen zijn: namen, getallen, lettertekens, tekst, haakjes en commentaar. (Omdat de Engelse termen veel gebruikelijker zijn, noemen we ze ook: identifiers, numeric literals, character literals, strings, delimiters, comments.) *Namen* bestaan uit een letter gevolgd door nul of meer letters, cijfers of onderstrepingstekens (dit laatste om de leesbaarheid te vergroten). Namen kunnen ook door Ada gereserveerde woorden zijn; daarvan zijn er 63, zie tabel 6-2. Namen mogen niet langer zijn dan één programmaregel. Dit zijn bijvoorbeeld toegelaten namen in Ada:

Kleuren_van_de_regenboog
temperatuur_voeler_37
Pagina_teller
POLL_TERMINALS

Tabel 6-2: Gereserveerde woorden in Ada

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array			pragma	then
at	else		private	type
	elsif	limited	procedure	
	end	loop		
begin	entry		raise	use
body	exception		range	
	exit	mod	record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

Ada maakt geen verschil tussen hoofdletters en kleine letters. De namen PAGINA_TELLER en pagina_teller stellen dus dezelfde grootheid voor; PAGINATELLER is een andere grootheid, want het onderstrepingsteken heeft wel degelijk betekenis. We zullen vanaf nu afspreken om gereserveerde woorden steeds in kleine letters te schrijven en door de gebruiker gecreëerde namen in hoofdletters. Commentaar laten we steeds voorafgaan door: --.

Getallen stellen geheeltallige of reële waarden voor. Getallen kunnen in elk talstelsel van tweetallig tot en met zestientallig worden voorgesteld. Ook in getallen mag het onderstrepingsteken worden gebruikt om de leesbaarheid te vergroten; de compiler negeert dit teken. Dit zijn toegelaten geheeltallige waarden:

7
1_000_000 -- hetzelfde als 1000000
1_00_00_00 -- dezelfde waarde
1e6 -- dezelfde waarde, te lezen als $1*10^6$
2#1100# -- binair equivalent van het decimale getal 12
16#C# -- hexadecimaal equivalent van het decimale getal 12

En dit zijn de toegelaten voorstellingswijzen van reële getallen:

0.125
3.141_592_65 -- een benadering voor π
2.78e-3 -- equivalent met 0.00278
1.0e6 -- equivalent met 1000000.0
16#F.0# -- equivalent met het decimale getal 15.0

Bij reële getallen moet èn voor èn na de decimale punt minstens één cijfer staan.

Een *letterteken* is één van de 95 ASCII-tekens tussen accenten. Een *tekst* of *string* is een rij van nul of meer tekens tussen aanhalingstekens. Voorbeelden:

'A' -- een letterteken
'*' -- nog eentje
''' -- het teken '
"" -- de lege string
"Dag van de week" -- een string ter lengte 15

Met *haakjes* of *delimiters* bedoelen wij de volgende verzameling symbolen:

' ( ) * + , - . / : ; < = > | &

en de samengestelde symbolen:

=> .. ** := /= >= <= << >> <>

Deze symbolen hebben speciale contextafhankelijke betekenissen.

Naburige lexicografische eenheden kunnen gescheiden worden door een of meer spaties. De eenheden moeten op één regel passen, maar hun plaats op de regels is volkomen vrij in Ada. *Commentaren* beginnen steeds met -- en zijn afgelopen aan het einde van de regel.

Er is nog één andere elementaire grootheid, die niet beschouwd wordt als een lexicografische eenheid. Een *pragma* is een aanwijzing voor de Ada compiler. De standaard in Ada opgenomen pragma's worden in appendix E beschreven.

Type definities en objectdeclaraties

Elke programmeertaal heeft:

- Regels voor het weergeven van de taal zelf.
- Objecten en mechanismen om die te definiëren (datatypen).
- Operaties en mechanismen om die te construeren (subprogramma's.

In de vorige paragraaf gaven we enige regels voor het weergeven van elementen uit de taal en nu concentreren we ons op datatypen en objecten. Een *datatype* bestaat uit:

- Een waardenverzameling.
- Een verzameling van op die waarden toegelaten bewerkingen.

Een datatype kan bijvoorbeeld zijn de grootheid WACHTRIJ. Een wachtrij bestaat uit een verzameling waarden (de wachtenden) en een stel bewerkingen (toevoegen aan de wachtrij, verwijderen uit de rij). Merk op dat het type WACHTRIJ slechts een blauwdruk of sjabloon is. Als we het hebben over 'de rij bij kassa drie bij de supermarkt om de hoek', dan hebben we het over een object van het type WACHTRIJ. Het object kan een bepaalde waarde hebben, zoals 'er zijn nog vier wachtenden voor u'.

In Ada zijn een aantal klassen van datatypen beschikbaar:

- *scalaire datatypen*
  integer (geheeltallig)
  real (reëel)
  enumeratie (opsomming)
- *samengestelde datatypen*
  array
  record
- *access datatypen*
- *privé datatypen*
- *subtype en afgeleide typen*

De eerste klasse (integers, reals en enumeraties) zijn scalairen: zij hebben niet meerdere samenstellende delen. *Integer typen* definiëren verzamelingen van geheeltallige waarden (geen decimalen).

Reële typen definiëren niet-gehele waarden (getallen in drijvende of vaste puntnotatie). Met behulp van *enumeratie typen* kan de gebruiker zijn eigen waardenverzamelingen specificeren. Voorbeelden van integer typen:

```
INTEGER     -- een tevoren gedefinieerd interval
NATURAL     -- idem, doch groter dan nul
type INDEX is range 1..50;
            -- een door de programmeur gedefinieerd interval
```

Voorbeelden van reële typen:

```
FLOAT                       -- een tevoren vastgesteld interval
type MASS is digits 10;     -- een type in drijvende punt notatie
type VOLTAGE is delta 0.01 range -12.0.. +24.0;
                            -- een vaste punt notatie
```

Voorbeelden van enumeratie typen:

```
BOOLEAN          -- voorgedefinieerd (FALSE,TRUE)
CHARACTER        -- ook voorgedefinieerd
type KLEUR is (ROOD,ORANJE,GEEL,GROEN,BLAUW,INDIGO,
               VIOLET);
                 -- door de gebruiker gedefinieerd type
```

Ada kent ook twee gestructureerde typen: array en record. *Arrays* zijn collecties van elementen van hetzelfde type, *records* kunnen collecties zijn van elementen van dezelfde of van verschillende typen. Voorbeelden van array definities:

```
type SCHAAKBORD is array (1..8,1..8) of SCHAAKSTUK;
                              -- tweedimensionaal array
type PIXEL is array (KLEUR) of FLOAT;
type SENSOR is array(INDEX range 5..10) of VOLTAGE;
```

Voorbeelden van toegelaten record type definities:

```
type DATUM is      -- volgt een record van drie componenten
  record
    DAG   : INTEGER range 1..31;
    MAAND : INTEGER range 1..12;
    JAAR  : NATURAL;
  end record;
type KLEP is
  record
    NAAM            : STRING(1..20);
    PLAATS          : STRING(1..30);
    OPEN            : BOOLEAN;
    STROOMSNELHEID  : FLOAT range 0.0 .. 30.0;
  end record;
```

De typen die we tot nu toe behandelden zijn statische typen: op het moment van compilatie zijn ze volledig bekend. In veel gevallen moeten objecten dynamisch (tijdens uitvoeren van het programma) worden gecreeërd. Denk bijvoorbeeld aan een systeem dat metingen doet en de frequentie van de metingen laat afhangen van de gemeten waarden. In dit geval kan niet van tevoren worden voorspeld hoeveel gegevens moeten worden opgeslagen. In Ada verwijzen access of toegangswaarden naar andere objecten. Zij kunnen gebruikt worden voor het dynamisch creëren van objecten:

```
type BUFFERPOINTER is access BUFFER;
                                -- toegang tot BUFFER objecten
```

Ook taken kunnen als type worden benoemd en dat betekent dat tijdens het uitvoeren van het programma nieuwe taken kunnen worden gecreeërd. In hoofdstuk 16 gaan we daar nader op in.

Een volgende klasse van typen in Ada zijn de *privé typen*, die alleen binnen pakketten worden gebruikt. Ook deze typen definiëren een waardenverzameling en een verzameling toegelaten operaties, maar bij privé typen is het mechanisme niet zoals bij andere typen voor de gebruiker toegankelijk. De programmeur kan bepaalde operaties voor privé typen definiëren en dit zijn dan de enige operaties die de gebruiker van het pakket kan toepassen. Op die manier geeft Ada de mogelijkheid de werking ontoegankelijk te maken en zo een nieuw niveau van abstracte datatypen te creëren. Zo kan bijvoorbeeld een stack of stapelmechanisme worden gedefinieerd:

```
package STACK is          -- het pakket dat een privé type omvat
  type BUFFER is private;
  procedure PUSH (ELEMENT: in INTEGFR; OP: in out BUFFER);
  procedure POP  (ELEMENT: out INTEGER; VAN: in out BUFFER);
private
  MAXIMUM_ELEMENTEN: constant INTEGER := 100;
  type LIJST is array (INTEGER range 1..MAXIMUM_ELEMENTEN)
                of INTEGER;
  type BUFFER is
   record
     STRUCTUUR: LIJST
     TOP        : INTEGER range 1..MAXIMUM_ELEMENTEN;
   end record;
end STACK;
```

Elk pakket moet een romp hebben ('body' in Ada), maar in dit voorbeeld laten we die weg. Elementen zijn nu van het type STACK.BUFFER en die enige operaties op die elementen zijn PUSH (iets op de stack zetten) en POP (iets van de stack halen). De structuur voorkomt dat de gebruiker direct toegang heeft tot BUFFER.

Ada heeft ook een mogelijkheid om typen verder onder te verdelen, namelijk via *subtypen* en *afgeleide typen*, zie verder hoofdstuk 8.

Datatypen geven de mogelijkheid data te structureren. Verschijningsvormen (objecten) van een bepaald type moeten met behulp van *declaraties* worden gecreëerd. Met behulp van objectdeclaraties kunnen in Ada variabelen en constanten worden gevormd. Dit zijn voorbeelden van declaraties van variabelen:

```
TELLER      : INTEGER;
                    -- gebruik van een voorgedefinieerd type
VERJAARDAG : DATUM;
                    -- een gebruikersgedefinieerd type
MIJN_BUFFER : STACK.BUFFER;
                    -- ook door de gebruiker gedefinieerd
```

Een dit zijn voorbeelden van constanten:

```
PI    : constant := 3.14159265;                -- een constant object
ADRES : constant INTEGER := 8#777_776#; -- een octale constante
```

De taal kent sterke typen, dat betekent dat we objecten van verschillende typen niet direct in bewerkingen mogen combineren. Ook dit draagt weer bij aan het opleggen aan de gebruiker van het eenmaal gekozen abstractieniveau.

Namen en expressies

In elke taal worden namen gebruikt om grootheden van een bepaald type te identificeren. In een groot programma kunnen honderden of zelfs duizenden verschillende namen voorkomen. Het kan dan moeilijk worden voor de programmeur om steeds nieuwe namen te verzinnen en om te voorkomen dat hij een naam dubbel gebruikt. Ada kent in dit verband het principe van *overloading* (voor deze technische term is geen Nederlands equivalent beschikbaar). Het overloading principe maakt het mogelijk dezelfde naam voor verschillende grootheden te gebruiken, zolang hierdoor tenminste geen tegenstrijdigheden ontstaan. Een voorbeeld met enumeratie typen:

```
type VOELER is (TEMPERATUUR,VOCHTIGHEID,DRUK);
                        -- een enumeratie type
type ALARM  is (NORMAAL,TEMPERATUUR,INBRAAK);
                        -- overloading van TEMPERATUUR
```

Overloading moet om onduidelijkheden te voorkomen spaarzaam worden toegepast.

Namen en constanten kunnen in expressies worden gebruikt om een waarde te berekenen (zie hoofdstuk 11). Ada beschikt over de gebruikelijke operatoren, hier opgesomd naar afnemende prioriteit:

```
**        not  abs            -- machtsverheffen en andere opera-
                              -- toren met hoogste prioriteit
*    /    mod  rem            -- vermenigvuldigingsoperatoren
+    -                        -- unaire optelling
+    -    &                   -- binaire optelling
=    /=   <    <=   >   >=    -- relationele operatoren
and  or   xor                 -- logische operatoren
and then          or else
```

Met deze operatoren kunnen expressies worden gevormd:

```
PI                  -- een eenvoudige expressie
(B**2)-(4.0*A*C)    -- een ingewikkelder expressie
CHAR in 'A'..'Z'    -- een BOOLEAN (logische) expressie
(2.789**4)+36.0     -- een statische expressie
```

In hoofdstuk 10 zullen we zien dat de betekenis van operatoren in Ada kan worden gewijzigd, zodat ze ook bij abstracte datatypen kunnen worden toegepast.

Instructies

Ada heeft niet alleen tal van mogelijkheden om data te beschrijven, maar ook een krachtige verzameling instructies, waarmee algoritmen kunnen worden geformuleerd. Omdat Ada een gestructureerde taal is, bezit Ada instructies voor sequentieel, iteratief en voorwaardelijk uitvoeren van opdrachten. Daarbij komen dan nog speciale instructies voor het besturen van taken en uitzonderingsgevallen. Een korte opsomming van instructies in Ada:

- *sequentiële besturing*
  waardetoekenning
  block
  null
  return
  subprogramma-aanroep
- *voorwaardelijke besturing*
  case
  if
- *iteratieve besturing*
  exit
  loop
- *andere instructies*
  abort
  accept
  code
  delay
  entry call
  goto
  raise
  select

De instructies worden in de hoofdstukken 10, 11, 16 en 17 uitgebreid besproken.

De waardetoekenningsopdracht en de subprogramma aanroep zijn de belangrijkste sequentiële besturingsstructuren. Met behulp van de *waardetoekenning* (*assignment*) wordt een berekende waarde aan een variabele toegevoegd:

```
TELLER := TELLER + 1;
                        -- een eenvoudige waardetoekenning
GEBOORTEDATUM.JAAR := 1955;
                        -- toekenning aan een component van een record
```

Met behulp van *subprogramma's* kunnen algoritmen die van een naam zijn voorzien in werking worden gesteld. In hoofdstuk 10 zullen we nader ingaan op de mogelijkheid aan de formele parameters actuele waarden toe te kennen met behulp van een positionele, maar ook met behulp van een 'naam' notatie:

```
VUL_TANK                              -- een procedure-aanroep
WAARDE := MATH_FUNC.TAN(HOEK);        -- een functie-aanroep
ROTEER(PUNTEN,30.0); -- gebruik van positionele notatie
ROTEER(PUNTEN,HOEK => 17.6);
                        -- gebruik van 'naam' notatie
```

Er zijn nog andere sequentiële instructies, maar de behandeling daarvan stellen we uit tot hoofdstuk 11.

De *if* en de *case* structuur worden in Ada voor voorwaardelijke besturing gebruikt. Met behulp van de *case* instructie wordt een alternatief uit een aantal mogelijkheden gekozen, afhankelijk van een expressie met een aftelbaar aantal mogelijke waarden. De *if* instructie selecteert nul of één mogelijkheid, afhankelijk van de uitkomst van een logische expressie. Voorbeelden:

```
if BUFFERLENGTE = MAXIMUM_BUFFERLENGTE then
  VERWERK_OVERFLOW;
end if;

if KLEPSTATUS = OPEN then
  LEES_STROOM(SNELHEID);
else
  LEES_DRUK(WAARDE);
end if;                     -- een if-then-else structuur

case BUFFERLENGTE is -- selectie uit meer dan een mogelijkheid
  when MAX_BUFFERLENGTE/2 =>STUUR_OVERFLOW_WAARSCHUWING;
                            LEES_VOLGENDE_WAARDE;
  when MAX_BUFFERLENGTE   =>VERWERK_OVERFLOW;
  when others             =>LEES_VOLGENDE_WAARDE;
end case;

case PIXEL_KLEUR is
  when ROOD|GEEL|BLAUW    =>VERGROOT_INTENSITEIT;
  when BRUIN .. WIT       =>MAAK ZWART;
  when others             =>null;
end case;
```

Iteratie of herhaling wordt in Ada uitgevoerd met een van de varianten van de *loop* instructie. Ada kent een elementaire lus, een tellende lus en een voorwaardelijke of 'while' lus. Via een *exit* instructie kan de herhaling worden gestopt. Voorbeelden zijn:

```
loop                                -- een elementaire lus
  LEES_MODEM(SYMBOOL);
  exit when EINDE_ZENDPERIODE;
  TOON(SYMBOOL);
end loop;

for I in LIJST'INTERVAL             -- een tellende lus
  loop
    SOM := SOM + LIJST(I);
  end loop;

while GEGEVENS_BESCHIKBAAR          -- een while lus
  loop
    LEES(MIJN_FILE,INPUT_BUFFER);
    VERWERK(INPUT_BUFFER);
  end loop;
```

De overige instructies in Ada hebben te maken met taken en behandelen van uitzonderingssituaties; de behandeling daarvan stellen wij uit tot later in het boek.

Subprogramma's

Zoals we zagen bevat Ada een aantal elementaire typen en een mechanisme om abstracte datatypen te creëren (met behulp van privé typen). Wat betreft het formuleren van algoritmen is een analoge constructie mogelijk voor het creëren van abstracte operaties. Dit gebeurt via het Ada subprogramma, waarbij twee klassen zijn te onderscheiden:

- procedures
- functies

Ook subprogramma's hebben (evenals pakketten en taken) een specificatiegedeelte en een romp of 'body'. In de specificatie staan de naam van het subprogramma, eventuele formele parameters en in het geval van functies, de typen van de functieresultaten. De romp omvat een rij instructies die tesamen het mechanisme van het algoritme vormen. Een voorbeeld:

```
procedure ROTEER(PUNTEN: in out COORDINAAT;
                   HOEK in RADIALEN) is
  begin
    --
    -- rij instructies
    --
  end ROTEER;

function HASH(SLEUTEL: in ELEMENT) return HASH_WAARDE is
  begin
    --
    -- rij instructies
    --
  end HASH;

function "*"(X,Y: MATRIX) return MATRIX is
  begin
    --
    -- rij instructies
    --
  end "*";
```

De termen *in*, *out* en *in out* geven de modus of richting aan van de gegevensstroom met betrekking tot het subprogramma; functies kunnen alleen *in* parameters hebben. In het laatste voorbeeld is te zien hoe een operatorsymbool opnieuw wordt gedefinieerd om matrix-vermenigvuldiging mogelijk te maken. In hoofdstuk 10 wordt de structuur en toepassingsmogelijkheid van subprogramma's verder behandeld.

Pakketten

Een volgende elementaire programma-eenheid in Ada is het pakket. Pakketten bieden de mogelijkheid een verzameling logisch samenhangende grootheden als het ware te verpakken. Op deze manier ondersteunen pakketten de principes van gegevensabstractie en beperkte toegankelijkheid. Vooral in hoofdstuk 13, maar ook in alle andere nu nog volgende hoofdstukken, zullen een aantal verschillende toepassingsmogelijkheden van pakketten worden behandeld. Net als subprogramma's hebben pakketten een specificatiegedeelte en een romp. De specificatie is de formulering van het contract dat de programmeur sluit met de gebruiker over de manier waarop het pakket moet worden gebruikt. De romp van het pakket krijgt de gebruiker nooit te zien. Hoe de functies in het nu volgende voorbeeld precies werken hoeft de gebruiker niet te weten:

```
package COMPLEX is  -- pakket voor rekenen met complexe getallen
  type GETAL is
    record
      REEEL_DEEL      : FLOAT;
      IMAGINAIR_DEEL: FLOAT;
    end record;
  function "+"(A,B: in GETAL) return GETAL;
  function "-"(A,B: in GETAL) return GETAL;
  function "*"(A,B: in GETAL) return GETAL;
end COMPLEX;
```

De bouwer van het pakket kan de body bijvoorbeeld zo vullen:

```
package body COMPLEX is        -- de implementatie
  function "+"(A,B: in GETAL) return GETAL is
    begin
      -- rij instructies
    end "+";
  function "-"(A,B: in GETAL) return GETAL is
    begin
      -- rij instructies
    end "-";
  function "*"(A,B: in GETAL) return GETAL is
    begin
      -- rij instructies
    end "*";
begin
  -- rij instructies voor toekennen beginwaarden
end COMPLEX;
```

Zowel de tekst als de instructies zijn voor de gebruiker ontoegankelijk. In hoofdstuk 13 zullen we ook laten zien hoe de structuur van het type COMPLEX.GETAL eveneens voor de gebruiker ontoegankelijk kan worden gemaakt, waarmee de abstractie op een nog hoger niveau wordt gebracht.

Taken

De laatste te behandelen categorie van Ada programma-eenheden wordt gevormd door de taken. Alle structuren die we tot nu toe behandelden waren bedoeld voor volgtijdelijke of sequentiële uitvoering. Vooral bij 'embedded' computertoepassingen moeten vaak een aantal acties tegelijkertijd worden uitgevoerd. In de procesbesturing is het vaak nodig dat een systeem een groot aantal verschillende meetinstrumenten controleert en onmiddellijk actie onderneemt als ergens een grenswaarde wordt overschreden. Indien de meetwaarden achtereenvolgens worden opgevraagd ('polling') dan bestaat steeds de kans dat zo'n extreme waarden juist wordt gemist. In dit geval wil de bouwer van het systeem bij voorkeur de

gelijktijdigheid van de werkelijkheid ook in het systeem tot uitdrukking kunnen brengen. Ada biedt de mogelijkheid hiertoe omdat in Ada een aantal parallelle taken kunnen worden gedefinieerd.

Taken in Ada zijn gebaseerd op het idee van communicerende processen. De binnen de taken uitgevoerde bewerkingen verlopen tegelijkertijd en onafhankelijk van elkaar, terwijl de communicatie door het uitwisselen van berichten tussen de taken wordt verzorgd. De uitwisseling van een bericht tussen twee taken heet een *rendezvous*, een ontmoeting dus. De communicatie wordt verzorgd door *entry* en *accept* instructies. Stel we hebben een taak die geregeld metingen doet van de spanning op een stroomdraad en die vervolgens, nadat een aantal metingen zijn uitgevoerd, een bericht stuurt naar een tweede taak. De aanroepende taak (METER) bereikt dan via *enter* de tweede taak (VERWERKER):

```
task body METER is          -- romp van de meettaak
begin
  loop
    -- instructies voor het meten van de spanning
    VERWERKER.RAPPORTAGE(METINGEN);
  end loop;
end METER;

task body VERWERKER is    --romp van de ontvangende taak
begin
  loop
    -- rij instructies
    accept RAPPORTAGE(METINGEN: in SPANNINGEN)
      do
        -- rij instructies
      end RAPPORTAGE;
    -- rij instructies
  end loop;
end VERWERKER;
```

Hoe de aangeroepen taak de metingen precies verwerkt zal meestal voor de aanroepende taak verborgen blijven. Het enige toegankelijke deel is de specificatie, die bijvoorbeeld kan luiden:

```
task VERWERKER is      -- specificatie van de taak
  entry RAPPORTAGE(METINGEN: in SPANNINGEN);
end VERWERKER;
```

Als de ene taak eerder aan communicatie toe is dan de andere, dan zal deze eerste taak wachten (*'inslapen'*) totdat ook de andere taak aan het rendezvous toe is. In hoofdstuk 16 zullen andere voorwaardelijke communicatiemogelijkheden worden behandeld:

- uitstellen (delay)
- selectieve communicatie

- selectief wachten
- tijdafhankelijke communicatie

Opvangen van uitzonderingen

Het is in veel gevallen belangrijk dat een systeem zich uit zichzelf snel herstelt van een situatie waarin een fout is opgetreden. In veel programmeertalen veroorzaken fouten, zoals het invoeren van letters als een getal werd verwacht, of bijvoorbeeld een deling door nul, dat het programma zichzelf 'ophangt'. Bij 'embedded' systemen, die volkomen onafhankelijk van menselijk ingrijpen gedurende lange tijd moeten kunnen draaien, is dit onacceptabel. Ada is daarom voorzien van een mechanisme voor het opvangen van fouten en uitzonderingssituaties. Een uitzonderingssituatie (het hoeft niet altijd een fout te zijn) kan ofwel tevoren zijn gedefinieerd (deling door nul), ofwel door de gebruiker worden omschreven (buffer overflow). Tevoren gedefinieerde uitzonderingssituaties kunnen zonder nadere omschrijving worden verwerkt; gebruikersgedefinieerde situaties moeten expliciet worden beschreven:

```
declare          -- start van een lokaal blok
  OVERVERHIT: exception;
begin
  loop
    LEES(TEMPERATUUR);
    SCHAAL(TEMPERATUUR);
    if TEMPERATUUR > LIMIET then
      raise OVERVERHIT;
    end if;
  end loop;
exception        -- behandeling van uitzondering
  when OVERVERHIT =>
    -- rij instructies
  when NUMERIEKE_FOUT =>
    -- rij instructies
end;
```

Als er een uitzonderingssituatie optreedt wordt de normale verwerking onderbroken en wordt de besturing door de 'exception handler' overgenomen. Is geen behandeling gedefinieerd dan wordt de besturing teruggegeven aan een volgend hoger niveau, totdat een definitie wordt gevonden, of totdat het niveau van besturingssysteem (operating system) wordt bereikt. Zie in dit verband ook hoofdstuk 17.

Generieke programma-eenheden

Ada werd ontworpen voor grote softwareprojecten en hulpmiddelen om de complexiteit te beheersen waren dus noodzakelijk. Een van die hulpmiddelen noemden we al: de mogelijkheid om eenheden afzonderlijk te compileren. Een tweede hulpmiddel is het gebruik van generieke programma-eenheden, die kunnen worden gebruikt als hetzelfde algoritme moet worden toegepast op verschillende datatypen. Aan het gemeenschappelijke algoritme kunnen de data items, waarop het kan worden toegepast, als parameters worden meegegeven en dit gebeurt tijdens de compilatie. We maken van het eerder ontwikkelde STACK pakket een generieke eenheid:

```
generic
   LIMIET: NATUURLIJK_GETAL
   type DATA is private;
package STACK is
   type BUFFER is private;
   procedure PUSH(ELEMENT: in  DATA; ON   : in out BUFFER;
   procedure POP (ELEMENT: out DATA; FROM: in out BUFFER;
private
   type LIJST is array(INTEGER range 1..LIMIET) of DATA;
   type BUFFER is
     record
       STRUCTUUR: LIJST;
       TOP      : INTEGER range 1..LIMIET;
     end record;
end STACK;
```

Een generieke definitie creëert geen uitvoerbare programmacode: het is een blauwdruk voor het algoritme. De gebruiker moet een 'verschijning' van de generieke eenheid creëren. Zo kunnen verschillende stacks worden gemaakt:

```
package INTEGER_STACK is new STACK(100,INTEGER);
package FLOAT_STACK   is new STACK(LIMIT=> 300,
                                   DATA=> FLOAT);
```

In het tweede pakket zijn de parameters weer benoemd om de leesbaarheid te bevorderen.

Specificatie van voorstellingswijzen

Ook als we in een hogere programmeertaal programmeren is het soms zinvol van machine-afhankelijke eigenschappen gebruik te maken. In andere programmeertalen kan dat meestal alleen via assembler subroutines, die vanuit het hoofdprogramma moeten worden aangeroepen. In Ada kunnen machine-afhankelijke eigenschappen en voorstellingswijzen van gegevens direct worden gebruikt. Dat kan via:

- adressering
- enumeratie van typerepresentatie
- lengtespecificatie
- record type specificatie

We geven hier maar een tweetal voorbeelden en verwijzen voor verdere behandeling naar hoofdstuk 17:

```
for GRADEN use 3*BYTES;                    -- lengtespecificatie
for PRINTER_STATUS use at 16#177_776#;     -- adresspecificatie
```

Input/output

Ingebouwde computersystemen zijn meestal niet direct aangesloten op de gebruikelijke apparatuur voor het invoeren en weergeven van gegevens, zoals printers en beeldschermen. Om met allerlei vreemde apparatuur te kunnen communiceren beschikt Ada over een aantal voorgedefinieerde pakketten voor:

- *hogere Input/Output*
  binaire I/O
  tekst I/O
- *lagere I/O*

De programmeur kan natuurlijk ook zijn eigen I/O pakketten definiëren, zie hiervoor verder hoofdstuk 19.

6.4 Samenvatting Van Taalbijzonderheden





Dit korte overzicht laat al zien dat Ada een uitgebreide, algemeen toepasbare hogere programmeertaal is. Ada kan helpen bij het ontwikkelen van betrouwbare en onderhoudbare programma's en Ada kan gebruikt worden voor grote systemen die vaak zullen moeten worden gewijzigd. Wellicht het belangrijkste van Ada is, dat de taal is gebaseerd op moderne software-ontwikkelingsmethodes en dus een geschikt instrument is bij het beheersen van de complexiteit van grote systemen. In de volgende hoofdstukken zullen we de vorm en het gebruik van de mogelijkheden van Ada in detail gaan bekijken.

Oefeningen

1. Het onderling vergelijken van de mogelijkheden van twee verschillende programmeertalen is weinig zinvol: zij kunnen best voor twee geheel verschillende toepassingsgebieden ontworpen zijn. Vanaf een hoger niveau van abstractie, zoals het niveau van dit hoofdstuk, heeft het echter wel zin Ada te vergelijken met talen als FORTRAN, COBOL, Pascal of een andere hogere programmeertaal. Vergelijk de mogelijkheden van de taal waarmee u het best bekend bent eens met die van Ada.

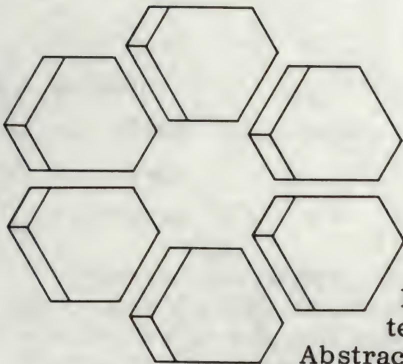
2. Welke eigenschappen van Ada ondersteunen direct de principes van abstractie en beperkt toegankelijk maken van gegevens (information hiding)? Zijn dat modulariteit en lokalisatie? Of zijn dat uniformiteit, volledigheid en testbaarheid?

3. Ada is een uitbreidbare taal. Dat betekent dat het mogelijk is onze eigen objecten en operaties te definiëren en te creëren. Welke tot nu toe behandelde eigenschappen van Ada bieden deze mogelijkheden?

4. Welke kenmerken van Ada dragen direct bij tot de leesbaarheid van de taal?

Pakket 3

DATASTRUCTUREN



Bij het verkrijgen van begrip voor ingewikkelde verschijnselen is het krachtigste gereedschap dat ons mensen ter beschikking staat: abstractie. Abstractievermogen komt voort uit het herkennen van overeenkomsten tussen bepaalde objecten, situaties of processen in de wereld om ons heen, en uit de beslissing ons te concentreren op die overeenkomsten en de verschillen tijdelijk te negeren.

C.A.R. Hoare
Notes on Data Structuring [1]

7 EERSTE ONTWERPPROBLEEM: TELLEN VAN BLAADJES

U weet nu iets van de ontwikkelingsgeschiedenis van Ada en u hebt een eerste idee gekregen van de kracht en de mogelijkheden van de taal. Vanzelfsprekend mag niet verwacht worden dat u in dit stadium al een volledig inzicht hebt in alle details, maar hopelijk hebt u wel al een globaal overzicht van de taalstructuur. In de nu volgende vijftien hoofdstukken zullen wij Ada nader bestuderen vanuit het gezichtspunt van de software-ontwikkelingsmethodes die wij in hoofdstuk vier en vijf introduceerden. We zullen daarbij uitgaan van vijf ontwerpvoorbeelden, die tesamen een aantal verschillende toepasssingsgebieden beslaan:

- tellen van blaadjes — hoofdstuk 7
- benaderen van een database — hoofdstuk 9, 12
- een pakket voor het werken met verzamelingen — hoofdstuk 15
- procesbesturing — hoofdstuk 18
- het 'kop op' beeldscherm — hoofdstuk 21

Bij elk probleem behandelen wij de analyse en het ontwerp. Hieruit volgt de formulering in Ada, waarbij we gebruik maken van onze object-gerichte ontwerpmethode. Deze benadering zal ons een top-down inzicht in de taal geven. Bij het voltooien van een oplossing zal blijken dat we nog andere taalgereedschappen nodig hebben en ook deze speciale mogelijkheden zullen we dan nader bestuderen. We willen echter zeker geen grabbelton met taalgereedschap presenteren; we zullen steeds de nadruk leggen op correct taalgebruik en voorbeelden geven van een programmeerstijl die helderheid en goed gebruik van de mogelijkheden van de taal benadrukt. Bij het bestuderen van dit gedeelte van het boek kunnen de appendices van pas komen: de Ada syntaxdiagrammen in appendix A, de stijlaanwijzingen in appendix B en de volledig uitgewerkte oplossingen van de ontwerpproblemen in appendix F.

We vatten de in hoofdstuk 5 behandelde ontwerpmethode nog eens samen:

- *Definieer het probleem.*

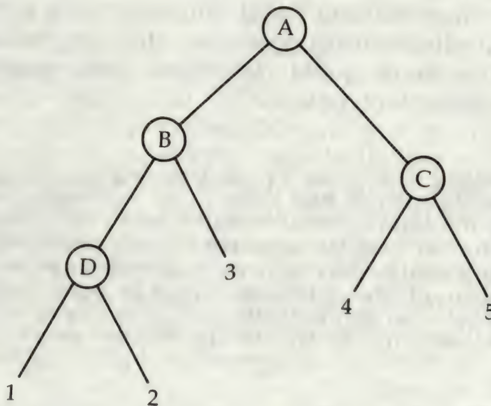
- *Ontwikkel een informele oplossingsstrategie.*
- *Formaliseer de strategie*
  identificeer de objecten en hun kenmerken
  identificeer de bewerkingen op de objecten
  stel de communicatiepatronen vast
  programmeer de bewerkingen.

We zullen deze methode nu gaan toepassen op ons eerste probleem: het tellen van de blaadjes in een binaire boom [1] (volgens de gewijzigde methode van Abbott). Dit lijkt een vrij academisch probleem om mee te beginnen, maar het zal dan ook het enige niet direct praktische probleem zijn dat we oplossen. Het probleem is ingewikkeld genoeg om niet triviaal te zijn, maar toch eenvoudig genoeg om te kunnen dienen als eerste voorzichtige stap naar het programmeren in Ada.

7.1 Definieer Het Probleem



Een *binaire boom* is een eenvoudige datastructuur, die veel gebruikt wordt in compilers, programma's die een spel tegen een (menselijke) tegenstander spelen en in database programmatuur. Figuur 7-1 geeft aan dat een binaire boom bestaat uit *knopen* (tussenliggende punten A, B, C, D) en *blaadjes* (de eindpunten 1, 2, 3, 4, 5). In een volledige binaire boom heeft elk knooppunt twee takken. In dat geval is ieder knooppunt ofwel een blaadje, ofwel er hangen twee binaire subbomen aan. Voor een boom die bestaat uit één blaadje geldt:



Figuur 7-1 Een binaire boom

```
BLAADJES(BOOM) = 1
```

Voor een boom die bestaat uit twee subbomen geldt:

```
BLAADJES(BOOM) = BLAADJES(RECHTER_SUBBOOM)
               + BLAADJES(LINKER_SUBBOOM)
```

Deze *recursieve* definitie blijft geldig, ook als een der subbomen leeg is (een nulboom). Wij willen nu een systeem ontwikkelen dat de blaadjes van een gegeven binaire boom telt.

7.2 Ontwikkel Een Informele Oplossingsstrategie



Het tellen van de blaadjes van een binaire boom kan op verschillende manieren gebeuren. We kunnen bijvoorbeeld bij een bepaald knooppunt beginnen en de boom bewandelen totdat alle blaadjes bezocht zijn. Dat is dan een imperatieve of gebiedende methode, waarvoor men tevoren moet weten hoe de knooppunten onderling fysiek verbonden zijn en dat is een implementatie afhankelijk gegeven. Hier zullen wij een algoritme toepassen die overeenkomt met een intuïtieve benadering. Op die manier kan de logische structuur van ons systeem onveranderd blijven, ook als de fysieke voorstellingswijze van de boomstructuur, bijvoorbeeld om redenen van efficiëntie, zou veranderen.

We gaan uit van de veronderstelling dat de taal waarin wij onze algoritme formuleren de drie elementaire besturingsstructuren (sequentie, voorwaarde en herhaling) kent, maar dat er geen van tevoren gedefinieerde objecten of operaties zijn. We veronderstellen ook dat de taal de mogelijkheid biedt tot uitbreiding via het definiëren en creëren van objecten en operaties daarop, zoals wij die nodig hebben voor ons abstracte model. Gegeven deze randvoorwaarden, volgt hier onze informele strategie:

> **Houdt een verzameling onderdelen van de boom bij die nog niet geteld zijn. Plaats om te beginnen de hele boom in die verzameling en ga er vanuit dat er nog nul eindpunten of blaadjes geteld zijn. Neem nu, zolang de verzameling niet leeg is, een boom uit de verzameling. Als de boom uit één blaadje bestaat, tel dat dan en gooi de boom weg. Bestaat de boom daarentegen uit twee subbomen, splits de boom dan in zijn linker en rechter subboom en plaats die twee bomen terug in de verzameling. Als de verzameling leeg is, dan zijn de blaadjes geteld. Toon dit aantal.**

In figuur 7-2 is een voorbeeld gegeven van deze informele strategie.

	BLAADJES	BOOM	VERZAMELING
1. Initieel:	0		A 1 B 2 3
2. Neem een boom uit de verzameling en bekijk die.	0	A 1 B 2 3	
3. Het is een boom: splits en zet subbomen terug.	0		B 2 3 1
4. Neem een boom uit de verzameling en bekijk die.	0	1	B 2 3
5. Het is een blaadje: tel het en gooi weg.	1		B 2 3
6. Neem een boom uit de verzameling en bekijk die.	1	B 2 3	
7. Het is een boom: splits en zet subbomen terug.	1		2 3
8. Neem een boom uit de verzameling en bekijk die.	1	3	2
9. Het is een blaadje: tel het en gooi weg.	2		2
10. Neem een boom uit de verzameling en bekijk die.	2	2	
11. Het is een blaadje: tel het en gooi weg.	3		
12. De verzameling is leeg: de blaadjes zijn geteld.	(3)		

Figuur 7-2 Voorbeeld van het tellen van blaadjes

7.3 Formaliseer De Strategie



De volgende stap is het formeel beschrijven van de informele strategie, waarbij we gebruik maken van Ada.

Identificeer de objecten en hun kenmerken

Dit is vrij eenvoudig, zoals we al zagen in hoofdstuk 5. We schrijven onze informele strategie nog eens op, maar deze keer onderstrepen we de zelfstandige naamwoorden en bijvoeglijke naamwoorden.

> **Houd een verzameling onderdelen van de boom bij die nog niet geteld zijn. Plaats om te beginnen de hele boom in die verzameling en ga er vanuit dat er nog nul blaadjes geteld zijn. Neem nu, zolang de verzameling niet leeg is, een boom uit de verzameling. Als de boom uit één blaadje bestaat, tel dat dan en gooi de boom weg. Bestaat de boom daarentegen uit twee subbomen, splits de boom dan in zijn linker en rechter subboom en plaats die twee bomen terug in de verzameling. Als de verzameling leeg is dan zijn de blaadjes geteld. Toon dit.**

Basisobjecten zijn dus:

- VERZAMELING
- BLAADJES
- BOOM, LINKER_SUBBOOM, RECHTER_SUBBOOM

Merk op dat BOOM, LINKER_SUBBOOM en RECHTER_SUBBOOM alle verschijningsvormen zijn van hetzelfde type, dat we BOOM_TYPE zullen noemen. Evenzo zijn er objecten van het BLAADJE_TYPE en het VERZAMELING_TYPE.

Identificeer de operaties op de objecten

Ook dit is niet moeilijk. We onderstrepen nu de werkwoorden en bijwoorden:

> **Houd een verzameling onderdelen van de boom bij die nog niet geteld zijn. Plaats om te beginnen de hele boom in die verzameling en ga er vanuit dat er nul nul blaadjes geteld zijn. Neem nu, zolang de verzameling niet leeg is, een boom uit de verzameling. Als de boom uit één blaadje bestaat, tel dat dan en gooi de boom weg. Bestaat de boom daarentegen uit twee subbomen, splits de boom dan in zijn linker en rechter subboom en plaats die twee bomen terug in de verzameling. Als de verzameling leeg is dan zijn de blaadjes geteld. Toon dit.**

Merk op dat het hier gaat om werkwoorden die duidelijk operaties

op de eerder onderscheiden objecten aangeven. 'Houd een verzameling bij' is geen operatie, maar geeft slechts het bestaan van het object VERZAMELING aan.

De operaties moeten nu aan de bijbehorende objecten worden toegekend. Dat moet ook gebeuren met bijwoorden of bijwoordelijke bepalingen, die bijvoorbeeld een tijdstip aangeven ('om te beginnen') of een voorwaarde ('niet leeg'). Dit leidt tot de volgende operaties op en eigenschappen van objecten:

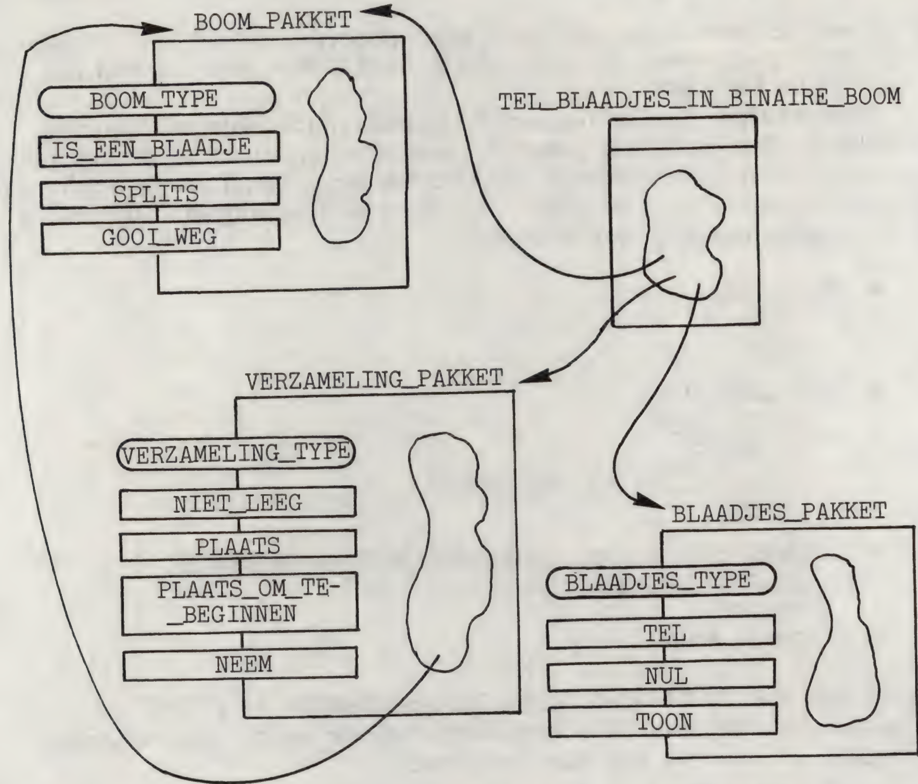
- BLAADJE
  TEL
  NUL
- VERZAMELING
  NIET_LEEG
  PLAATS
  PLAATS_OM_TE_BEGINNEN
  NEEM
- LINKER_SUBBOOM, RECHTER_SUBBOOM, BOOM
  IS_EEN_BLAADJE
  SPLITS
  GOOI_WEG

Omdat LINKER_SUBBOOM, RECHTER_SUBBOOM en BOOM verschijningsvormen van hetzelfde type zijn, hebben we de bijbehorende operaties ook slechts één maal genoemd.

Stel de communicatiepatronen vast

Nu we de objecten kennen en de operaties die erop kunnen worden uitgevoerd, kunnen we de relaties tussen de objecten beschrijven. In figuur 7-3 is de oplossing schematisch weergegeven met behulp van de in hoofdstuk 6 ontwikkelde symbolen. De belangrijkste actie in onze oplossing, het tellen van de blaadjes, wordt weergegeven als een subprogramma en de objecten worden vertegenwoordigd door drie pakketten. De pakketten bevatten de definities van de typen van de objecten en de specificaties van de toegelaten operaties. De namen van de typen en de operaties worden uit het pakket geëxporteerd. (Met *exporteren* wordt bedoeld: beschikbaar maken buiten het pakket.) De pijlen geven aan welke programma-eenheden toegankelijk zijn voor andere eenheden. Het hoofdprogramma kan alle drie pakketten zien, maar het omgekeerde is niet het geval. Het VERZAMELING_PAKKET kan echter wel het BOOM_PAKKET zien, want de VERZAMELING moet immers weten wat de aard is van de objecten die men PLAATSt of NEEMt.

Op grond van dit ontwerp kunnen we nu het communicatiepatroon of interface van elk pakket beschrijven. Het interface vormt een contract tussen het pakket en zijn gebruiker: de details blijven verborgen en slechts het communicatiepatroon is zichtbaar. We



Figuur 7-3 Ontwerp voor TEL_BLAADJES_IN_BINAIRE_BOOM

blijven verder de objecten vanaf het bijbehorende abstractieniveau beschouwen, omdat alleen de specificatie van de toegelaten operaties aan de gebruiker bekend is. Het onderliggende mechanisme van die operaties blijft voor de gebruiker verborgen. De gebruiker hoeft alleen maar in staat te zijn ze toe te passen.

In Ada worden de interfaces van de grootheden die in de oplossing voorkomen formeel gedeclareerd. De pakketspecificatie vormt het zichtbare deel van de communicatiemogelijkheid en de afzonderlijk compileerbare 'body' van het pakket bepaalt het mechanisme van de bijbehorende operaties. In hoofdstuk 13 zullen we de pakketstructuur in Ada verder behandelen. Op dit ogenblik is het voldoende te weten dat de pakketspecificatie onze abstracte datatypen definiëren.

Voor BLAADJE, een object van het type BLAADJES_TYPE wordt de interface:

```
package BLAADJES_PAKKET is
  type BLAADJES_TYPE is limited private;
  procedure TOON(BLAADJE: in      BLAADJES_TYPE);
  procedure TEL  (BLAADJE: in out BLAADJES_TYPE);
  procedure NUL  (BLAADJE: out    BLAADJES_TYPE);
private
  . . .
end BLAADJES_PAKKET;
```

In deze pakketspecificatie maakten we van de operaties procedures. De woorden **in**, **in out** en **out** zijn gereserveerde woorden en worden *modi* (enkelvoud: modus) genoemd. Zij geven de richting aan van de gegevensstroom ten opzichte van het subprogramma. Het BLAADJES_TYPE declareerden we verder als **limited private**, en dit geeft aan dat de structuur van dit type niet zichtbaar en dus niet bruikbaar is van buiten het pakket. De uitwerking van de private delen geven wij hier niet: in hoofdstuk 8 wordt ingegaan op de daarvoor benodigde primitieve typen in Ada.

VERZAMELING is een object van het type VERZAMELING_TYPE en de interface daarvan kan als volgt worden geformuleerd:

```
with BOOM_PAKKET
package VERZAMELING_PAKKET is
  type VERZAMELING_TYPE is limited private;
  function NIET_LEEG(VERZAMELING: in VERZAMELING_TYPE)
                                   return BOOLEAN;
  procedure PLAATS (BOOM: in out BOOM_PAKKET.BOOM_TYPE;
                    IN  : in out VERZAMELING_TYPE);
  procedure PLAATS_OM_TE_BEGINNEN
                   (BOOM: in out BOOM_PAKKET.BOOM_TYPE;
                    IN  : in out VERZAMELING TYPE);
  procedure NEEM   (BOOM: out    BOOM_PAKKET.BOOM_TYPE;
                    UIT : in out VERZAMELING_TYPE);
private
  . . .
end VERZAMELING_PAKKET;
```

In deze pakketspecificatie wordt ook weer een **private** gedeelte gebruikt. Dit zullen we hier niet verder uitwerken. Als een type **limited private** wordt gedefinieerd, dan zijn alleen de operaties die in de pakketspecificatie worden genoemd van buiten het pakket benaderbaar en te gebruiken. Zelfs het toekennen van een waarde aan een binnen het pakket gebruikte variabele of het vergelijken van twee waarden is voor de gebruiker van het pakket niet mogelijk. In de hier gebruikte programmeerstijl worden procedures gebruikt om de bewerkingen te benoemen, terwijl eigenschappen (resultaten van expressies, met als mogelijke waarden TRUE en FALSE) met behulp van functieprocedures worden beschreven.

Binnen het verzamelingspakket moet bekend zijn hoe de objecten eruit zien, die als elementen van de verzameling gebruikt mogen

worden. In dit geval moet het BOOM_PAKKET binnen het VERZAMELING_PAKKET te gebruiken zijn. Dit wordt mogelijk door de with-constructie: 'with BOOM_PAKKET' aan het begin van het VERZAMELING_PAKKET. Nu kunnen de elementen uit het BOOM_PAKKET via de puntnotatie, zoals bijvoorbeeld in: BOOM_PAKKET.BOOM_TYPE, gebruikt worden.

Tenslotte beschrijven we het BOOM_PAKKET zelf:

```
package BOOM_PAKKET is
  type BOOM_TYPE is private;
  function IS_EEN_BLAADJE(BOOM: in BOOM_TYPE)
                           return BOOLEAN;
  procedure SPLITS        (BOOM:    : in out BOOM_TYPE;
                           LINKER   :    out BOOM_TYPE;
                           RECHTER:      out BOOM_TYPE);
  procedure GOOI_WEG      (BOOM     : in out BOOM_TYPE);
private
  . . .
end BOOM_PAKKET;
```

Ook hier zien we af van verdere uitwerking van het private gedeelte. Bij enkele procedures gebruikten we bij de parameters de in out modus, en wel om de volgende reden. Bijvoorbeeld in de procedure SPLITS is de bedoeling de oorspronkelijke BOOM te nemen, in twee subbomen te verdelen en vervolgens als resultaat alleen die twee subbomen op te leveren en de oorspronkelijke boom te laten verdwijnen door er een 'nul-boom' van te maken. Als we dat laatste niet zouden doen en dus de oorspronkelijke boom niet als nul-boom zouden uitvoeren als 'out'-parameter, dan zou de splitsing niet correct verlopen.

Programmeer de bewerkingen

Nu we het BLAADJES_TYPE, het BOOM_TYPE en het VERZAMELING_TYPE hebben gespecificeerd, staat het gereedschap voor de oplossing van ons probleem tot onze beschikking. Nu kan de informele strategie in de programmeertaal geformuleerd worden. Verdiep u nog niet in de Ada taalelementen; de regels voor de taal zullen we later uitgebreid behandelen. Hier gaat het om de oplossing, die, omdat de formulering zo sterk overeenkomt met die van de probleemstelling, zeer leesbaar en begrijpelijk is:

```
with BLAADJES_PAKKET,VERZAMELING_PAKKET,BOOM_PAKKET;
use  BLAADJES_PAKKET,VERZAMELING_PAKKET,BOOM_PAKKET;
procedure TEL_BLAADJES_IN_BINAIRE_BOOM is
  BLAADJES          : BLAADJES_TYPE;
  LINKER_SUBBOOM    : BOOM_TYPE;
  VERZAMELING       : VERZAMELING_TYPE;
  RECHTER_SUBBOOM: BOOM_TYPE;
  BOOM              : BOOM_TYPE;
```

```
begin
  PLAATS_OM_TE_BEGINNEN(BOOM,IN => VERZAMELING);
  NUL(BLAADJES);
  while NIET_LEEG(VERZAMELING)
    loop
      NEEM(BOOM,UIT => VERZAMELING);
      if IS_EEN_BLAADJE(BOOM) then
        TEL(BLAADJES);
        GOOI_WEG(BOOM);
      else
        SPLITS(BOOM,
               LINKER => LINKER_SUBBOOM,
               RECHTER => RECHTER_SUBBOOM);
        PLAATS(LINKER_SUBBOOM,IN => VERZAMELING);
        PLAATS(RECHTER_SUBBOOM,IN => VERZAMELING);
      end if;
    end loop; TOON(BLAADJES);
end TEL_BLAADJES_IN_BINAIRE_BOOM;
```

De **with**-instructie wordt weer gebruikt om de pakketten te benoemen die vanuit de procedure zichtbaar moeten zijn. De **use**-instructie heeft een vrij subtiele betekenis, daarop zullen we in hoofdstuk 13 nader ingaan. Tesamen creëren deze twee instructies het verband tussen de pakketten, zoals dat in figuur 7-3 is aangegeven. Nadat de procedure is gedeclareerd,volgen de objecten, zoals we die bij het formuleren van de oplossing hebben onderscheiden. Tenslotte volgt na het **begin** van het blok, de uitwerking van het blaadjes-telalgoritme zelf. Onze informele strategie is volledig terug te vinden in de formulering in Ada.

Nu zouden de 'bodies' van de pakketten nog moeten worden uitgewerkt, en de daarin als 'private' gedeclareerde gedeelten. Op dit moment zijn echter nog niet genoeg Ada gereedschappen geïntroduceerd om dit mogelijk te maken. We hebben bijvoorbeeld een aantal abstracte datatypen ingevoerd, zoals BOOM_TYPE, en deze typen moeten nog op een lager abstractieniveau verder worden uitgewerkt. We sluiten daarom de behandeling van dit eerste probleem hier af en zullen hierna de mogelijkheden van Ada voor het beschrijven van gegevensstructuren behandelen.

Oefeningen

*1. Een oplossing voor het blaadjes-telprobleem zou ook recursief kunnen worden geformuleerd. Bekijk de recursieve definitie die in dit hoofdstuk werd gegeven en gebruik vervolgens de instrumenten uit het BOOM_PAKKET om de oplossing in Ada uit te schrijven.

2. Kan een gebruiker van de beschreven pakketten het abstractieniveau van de objecten op de een of andere wijze doorbreken? Licht uw antwoord toe.

3. Stel we wijzigen de voorstellingswijze van BOOM_TYPE, zijn er dan programma-eenheden in ons systeem die daardoor beïnvloed worden? Zo ja, welke?

4. Als u een algemeen BOOM_PAKKET zou moeten schrijven, dat alle mogelijke operaties op bomen moet bevatten, welke operaties zou u dan toegankelijk maken via de pakketspecificatie?

8 ABSTRACT VOORSTELLEN VAN GEGEVENS EN ADA'S DATATYPEN

Alle natuurlijke talen kennen zelfstandige naamwoorden en werkwoorden. Bijvoeglijke naamwoorden en bijwoorden worden slechts gebruikt ter versterking of beperking en zijn afhankelijk van de zelfstandig naamwoord/werkwoord constructie. De combinatie van deze taalelementen maakt communicatie mogelijk en helpt ons bij het denken.

Je zou dus verwachten dat programmeertalen een soortgelijke structuur hebben met iets als werkwoorden en zelfstandige naamwoorden, in de vorm van bewerkingen of operaties en objecten. De meeste hogere programmeertalen zijn echter nogal gebiedend van aard; ze gaan uit van bevelen tot acties. NEEM de gegevens, PLAATS iets op een stack, en ITEREER om een proces te herhalen. In hoofdstuk 4 zagen we al, dat de eerste en tweede generatie programmeertalen voornamelijk gericht waren op evaluatie van expressies en besturing van de volgorde van uitvoering. Bij de ontwikkeling van derde generatie talen werd ook rekening gehouden met het belang van de mogelijkheid om de structuur van gegevens te kunnen beschrijven. Erkend werd dat het ging om de combinatie van het afbeelden van objecten én mogelijke bewerkingen daarop; elke nieuwe taal zou daartoe de mogelijkheid moeten bieden. In dit hoofdstuk zullen we verder ingaan op het idee van de abstracte voorstellingswijze van gegevens en we zullen zien hoe Ada met behulp van mechanismen voor typedefinities deze abstracte voorstellingswijze mogelijk maakt.

8.1 Abstract Voorstellen Van Gegevens



In zekere zin zijn al onze waarnemingen van de wereld om ons heen abstracties. Zelfs de stoel waarop u zit is een abstractie; op een lager abstractieniveau is slechts sprake van een verzameling moleculen, die op zichzelf ook weer abstracties zijn. 'Stoel' is alleen maar een etiket dat we op een bepaalde verzameling plakken om gemakkelijker over de eigenschappen van die verzameling te kunnen praten.

Zo noemen we bijvoorbeeld een op een bepaalde wijze georganiseerde verzameling plastic, metaal, glas en rubber een 'auto'. Het abstracte idee 'auto' heeft echter andere eigenschappen dan zijn samenstellende delen afzonderlijk.

Onze abstracties staan niet op zichzelf, maar zijn samengesteld uit een aantal lagen. Als we ons naar een hoger abstractieniveau begeven, krijgen we een beter overzicht ('we zien het hele plaatje'), terwijl de details van de lagere niveau juist minder goed te onderscheiden zijn. We kunnen deze lagen van abstractie door middel van een soort ladder weergeven:

- atomen (een zeer primitief wereldbeeld)
- organen (verzameling atomen met biologische betekenis)
- Minet (naam van een specifieke verzameling organen)
- kat (klasse, waaruit Minet een verschijningsvorm is)
- dier (klasse, waaruit kat een der verschijningsvormen is)
- levend organisme (dieren, maar ook planten)
- essentie (een zeer hoog abstractieniveau)

Vanaf elke sport op deze ladder beschouwt men hetzelfde object op een ander abstractieniveau. De ladder zou in beide richtingen kunnen worden uitgebreid en er zouden ook nieuwe sporten kunnen worden tussengevoegd.

Uit dit voorbeeld zijn twee belangrijke aspecten van het abstractiebegrip af te leiden. Ten eerste bestaat er niet één 'correct' niveau van waaruit we de wereld om ons heen kunnen beschouwen. Een dierenarts bevindt zich meestal op het niveau van de organen; een filosoof kan zich op het niveau van de essentie bevinden. Het juiste abstractieniveau hangt dus af van de toepassing. Ten tweede wordt elk niveau opgebouwd uit de lagere niveaus. Organen bestaan uit atomen; een specifiek voorbeeld van een kat is Minet.

Het gaat bij het creëren van abstracties niet alleen maar om het beschrijven van zaken. Met behulp van abstracties wordt het mogelijk te spreken over de eigenschappen van een bepaalde klasse van grootheden. De eigenschappen van grootheden op een bepaald abstractieniveau kunnen niet eenvoudigweg worden gereconstrueerd uit de eigenschappen van de grootheden op de lagere niveaus. Het is bijvoorbeeld weinig zinvol de werking van een virtueel geheugen van een computer te gaan verklaren op moleculair niveau. Of denk eens aan het gebruik van geheeltallige of INTEGER grootheden. Binnen de computer worden INTEGERs opgebouwd met behulp van nullen en enen, maar bij het rekenen met gehele getallen is het zelden nodig naar dat binaire niveau af te dalen. We kunnen rustig werken met operaties als optellen, aftrekken, vermenigvuldigen en delen, zonder dat we telkens hoeven te bedenken dat deze bewerkingen zijn geconstrueerd met behulp van operaties op binaire grootheden, zoals 'shift', AND, OR en NOT.

Abstracties maken het mogelijk ons te concentreren op de hoofdzaken en de bijzaken te negeren. Als u een auto bestuurt zou het maar afleiden als u voortdurend zou nadenken over de chemische reacties die in de cylinders plaatsvinden; met dit implementatiedetail hoeft u zich niet bezig te houden.

We legden er al herhaalde malen de nadruk op dat de oplossingen die we met onze hardware- en softwaregereedschappen creëren, zoveel mogelijk een model moeten vormen van onze abstracte voorstellingswijze van de werkelijkheid. In het binaire boomprobleem uit hoofdstuk 7 ging het bijvoorbeeld om objecten als BLAADJES, BOOM, en VERZAMELING. We konden die objecten manipuleren zonder dat we ons hoefden te verdiepen in hun voorstellingswijze op een lager niveau; het ging immers alleen maar om de logische eigenschappen van de objecten. Op een lager niveau kan dan wel blijken dat deze objecten zijn gecreëerd met behulp van arrays of pointers, maar op elk niveau doen we alsof een object één geheel is en dus niet uit elementaire delen is opgebouwd.

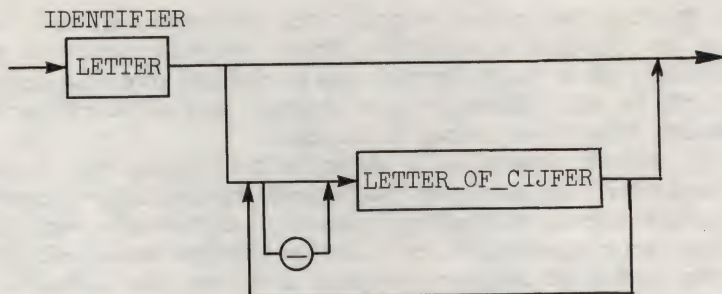
Bij eerste en tweede generatie talen zijn er maar weinig mogelijkheden voor het abstract voorstellen van gegevens. Je kunt natuurlijk wel een buffer maken van een array en een aantal pointers, maar bij het gebruik blijft het dan steeds nodig de stap te maken van de fysieke voorstellingswijze naar de logische betekenis op een hoger niveau. Ook is het in dit soort talen steeds mogelijk bewerkingen op de buffer uit te voeren, die in strijd zijn met de logische bufferstructuur. Zouden we bijvoorbeeld een wachtrij (een First In First Out of FIFO-buffer) creëren, dan zou FORTRAN noch COBOL kunnen voorkomen, dat we direct een waarde aan bijvoorbeeld het vijfde element van de buffer toekenden. Ook in Pascal is een dergelijke bescherming op logisch niveau niet mogelijk, hoewel Pascal wel meer mogelijkheden tot het structureren van gegevens biedt.

We eisen dus van onze programmeertaal dat:

- Gereedschap beschikbaar is om abstracte datastructuren te definiëren.
- Men gedwongen is om zich ook aan de eenmaal gedefinieerde eigenschappen te houden.

In het voorgaande hoofdstuk werd al aangegeven hoe het 'package'-mechanisme gebruikt kan worden om abstracte gegevensvoorstellingen te definiëren, en ook hoe het pakket kan dwingen tot het eerbiedigen van de gedefinieerde logische eigenschappen (*type encapsulation*). Elk hoger niveau van gegevensabstractie is echter opgebouwd met behulp van lagere datatypen, zoals INTEGERs of CHARACTERs. Deze primitieve datatypen zullen we nu nader gaan bekijken.

Daarvoor hebben we allereerst een hulpmiddel nodig om toegelaten constructies in Ada begrijpelijker te beschrijven. We gebruiken daartoe het *syntaxdiagram*. Een syntaxdiagram, zoals in figuur 8-1, is een soort spoorwegemplacement, waarmee elke toegelaten vorm van een bepaalde constructie in de taal kan worden gegenereerd.



Figuur 8-1 Syntaxdiagram voor IDENTIFIER

Dit syntaxdiagram beschrijft een Ada-naam of 'identifier'. Als u aan de linker kant bij de ingang begint en het spoor volgt tot u uiteindelijk bij de uitgang rechts uitkomt, dan heeft u een toegelaten naam in Ada geconstrueerd. In een cirkel of ellips weergegeven tekens, zoals de onderstreping '_' in figuur 8-1, komen precies zo in de taal Ada voor. Een rechthoek, zoals de rechthoek met het woord LETTER, bevat een grootheid die met een ander syntaxdiagram verder gedefinieerd wordt. Volgens het syntaxdiagram zijn de volgende namen in Ada toegelaten:

```
x
VERSNELLING
zwaartepunt
Knoop_77
V003
```

The volgende namen zijn niet toegelaten; zij kunnen niet met behulp van het syntaxdiagram worden gevormd:

```
4314
AFSTAND__TOT__BESTEMMING
KILOMETER-PER-UUR
AANTAL_#
x_
_x
```

Een syntaxdiagram geeft geen volledige informatie over de betekenis (*semantiek*) van een constructie. De regel, dat een naam niet langer mag zijn dan een programmaregel blijkt er bijvoorbeeld niet uit. In appendix A staan de syntaxdiagrammen voor alle Ada constructies. Onduidelijkheden over toegelaten formuleringen in Ada kunnen met behulp van deze diagrammen altijd worden opgehelderd.

8.2 Typen



In natuurlijke talen manipuleren we zelfstandige naamwoorden; in Ada worden die *objecten* genoemd. Elk object heeft een aantal eigenschappen (bij elkaar het *type* van het object genoemd) bestaande uit de waarden die het object kan aannemen en de bewerkingen die op het object kunnen worden uitgevoerd. Ada kent geen impliciete objecten (zoals bijvoorbeeld real en integer variabelen in FORTRAN), maar elke grootheid moet expliciet worden gedeclareerd. Voorbeelden:

```
COEFFICIENT   : FLOAT;
TELLER        : INTEGER;
NAAM          : STRING(1..80);
BOOM          : BOOM_TYPE;
WATER_OPSLAG  : TANK;
```

Elke grootheid in Ada moet gedeclareerd zijn, voordat hij kan worden gebruikt. Een declaratie bestaat uit het noemen van de naam van een object (zoals COEFFICIENT, TELLER, etcetera), en het vervolgens verbinden van het object met een bepaald type. Op objectdeclaraties zullen we later nader ingaan, hier is het voldoende een declaratie te beschouwen als de creatie van een verschijningsvorm met een naam van een bepaald type gegevens. Het datatype INTEGER verwijst bijvoorbeeld naar een verzameling toegelaten waarden en bewerkingen; TELLER is de naam van een object dat de eigenschappen heeft van een grootheid met het type INTEGER.

Het toekennen van een type biedt de mogelijkheid structuur aan gegevens op te leggen. Het expliciet toekennen van typen is een belangrijke eigenschap van Ada, vooral als we denken aan de in hoofdstuk 4 behandelde software-ontwikkelingsmethodes. Aan een aantal criteria voor goed ontwerp wordt door middel van het toekennen van datatypen voldaan [1]:

- *Onderhoudbaarheid*: beschrijven van objecten met een aantal eigenschappen.
- *Leesbaarheid*: expliciete omschrijving van die eigenschappen.
- *Betrouwbaarheid*: de garantie dat van die eigenschappen niet kan worden afgeweken.
- *Vereenvoudiging*: het ontoegankelijk maken van het onderliggende mechanisme.

Een type karakteriseert:

- Een waardenverzameling.
- Een verzameling operaties van toepassing op objecten van dit type.

Ada is een taal met *sterke typen*. Dit betekent dat objecten van een bepaald type *alleen* de voor dit type toegelaten waarden kunnen aannemen en dat er ook *alleen* maar de voor dit type gedefinieerde operaties op kunnen worden toegepast. Alle type informatie in Ada is statisch. Dit betekent dat bij de compilatie de typen van alle objecten bekend moeten zijn, tesamen met de waardenverzamelingen en de operaties. Op deze wijze geeft Ada een extra beveiliging, omdat met elkaar in tegenspraak zijnde type definities al tijdens de compilatie kunnen worden ontdekt. Dit betekent waarschijnlijk meer fouten tijdens de compilatie, maar vergroot tevens de kans op correctheid van het programma tijdens de uitvoering.

Het idee van sterke typen in ook terug te vinden in de ladder van abstractieniveaus. Op elk niveau zijn er toegelaten waarden en operaties. Om een voorbeeld te geven: de naam APPEL verwijst naar een type vrucht. Voor een kweker zijn de op APPEL toepasbare bewerkingen: planten, bemesten, plukken en rijpen. Het van elkaar aftrekken van APPELS of ze tot een macht verheffen heeft voor de kweker geen betekenis!

Met behulp van typen kan dus de structuur van gegevens worden aangegeven. Natuurlijk kan geen enkele taal zoveel basiselementen bevatten dat elk datatype, dat men maar bedenken kan, er direct in geformuleerd kan worden. Het idee is, een aantal elementaire eigenschappen in de taal op te nemen, met behulp waarvan datatypen kunnen worden geformuleerd, al naar gelang daar behoefte aan is. Ada kent vier klassen elementaire typen. Deze vormen de bouwstenen voor de typen van de objecten die bij een bepaald probleem horen. Deze vier klassen zijn:

- *Scalairen*: niet samengestelde waarden.
- *Composieten*: uit waarden van meerdere objecten samengestelde waarden.
- *Access-grootheden*: bieden toegang ('access') tot andere objecten.
- *Private-grootheden*: waarden zijn aan de gebruiker onbekend.

Deze klassen worden verder aangevuld met subtypen en afgeleide typen, waarmee een mogelijkheid tot verdere onderverdeling van de eigenschappen van grootheden wordt geboden. In hoofdstuk 16 worden *taak-typen* behandeld. Hun eigenschappen zijn te vergelijken met die van private-typen.

Voordat we nader op het werken met datatypen ingaan, nog twee opmerkingen ten aanzien van de wijze waarop in Ada typen worden behandeld. Ten eerste bevat Ada een aantal voorgedefinieerde typen, zoals INTEGER, FLOAT, BOOLEAN en CHARACTER. Deze datatypen en de bijbehorende operaties worden gedefinieerd in een machine-afhankelijk pakket, STANDARD genaamd, dat in appendix C is opgenomen. Telkens als een Ada programma-eenheid wordt gecompileerd, gebeurt dit binnen de context van het pakket STANDARD.

Dit betekent dat de daarin voorkomende datatypen automatisch ter beschikking van de gebruiker staan. Toch is het, en hier komen we later op terug, in het algemeen beter zelf onze numerieke typen te definiëren en niet gebruik te maken van de in Ada voorgedefinieerde typen.

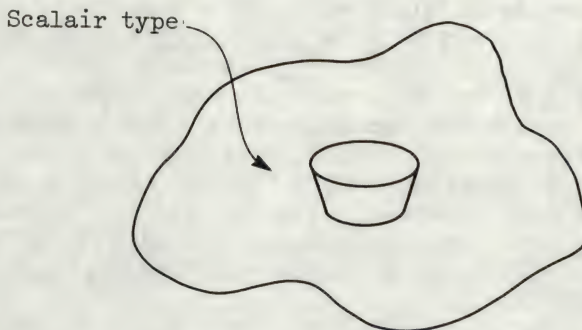
Binnen programma's is het soms nodig te verwijzen naar eigenschappen van een type, zoals het aantal CIJFERS van een numeriek type, de LENGTE van een array, of over de MACHINE_AFRONDING van een bepaalde computer bij numerieke benaderingen. Deze eigenschappen worden in Ada *attributen* genoemd. Attributen worden op een bepaalde standaardwijze aangegeven, en wel als volgt:

```
INTEGER'FIRST      -- de eerste INTEGER waarde
CHARACTER'SIZE     -- het aantal bits in CHARACTER objecten
NAME'ADDRESS       -- het geheugenadres van het object NAME
```

In appendix D worden alle voorgedefinieerde attributen vermeld en in de volgende paragrafen zullen er een aantal van vermeld worden.

Scalaire datatypen

Een *scalair type* bepaalt een verzameling niet samengestelde waarden. De vormeloze figuur in figuur 8-2 geeft aan dat een type zelf niets anders is dan een blauwdruk voor objecten. Het emmertje geeft aan dat scalaire objecten maar één waarde kunnen bevatten. Scalaire typen kunnen verder in drie categorieën worden onderverdeeld:




```
Waarden hebben geen componenten
  Geheeltallige (integer) typen
  Reële (real) typen
  Enumeratietypen
```

Figuur 8-2 Scalaire Datatypen

- geheeltallige (integer) typen
- reële (real) typen
- enumeratietypen

De integer en real typen heten *numerieke typen*. De integer en enumeratietypen worden ook wel *discrete typen* genoemd, omdat zij uit discrete (aftelbare) waarden bestaan en bijvoorbeeld voor indexering en tellen binnen lussen kunnen worden gebruikt. De waarden van scalaire typen zijn geordend.

Integer typen. Een *integer type* definitie voert een verzameling opeenvolgende gehele getallen in als waardenverzameling. Deze waarden kunnen door de computer exact worden voorgesteld; dit in tegenstelling tot reële getallen, die alleen maar bij benadering kunnen worden voorgesteld. De gehele getallen zijn waarden uit een niet begrensd type zonder naam, de *universele integer*. Deze laatste is het type van de gehele getallen in wiskundige zin. Dit type heeft geen naam en kan daarom vanuit programma's niet direct worden benaderd. Elke machine-implementatie definieert een deelverzameling van de gehele getallen in wiskundige zin.

De programmeur kan van de voorgedefinieerd integer typen uit het pakket STANDARD gebruik maken, bijvoorbeeld:

```
TELLER     : INTEGER;
BEVOLKING : LONG_INTEGER;
INDEX      : SHORT_INTEGER;
```

Het voorgedefinieerde INTEGER type (dat niet verward moet worden met de ruimere klasse van het Ada integer type) heeft een waardebereik afhankelijk van de machine-implementatie. Op een 16 bit machine kan INTEGER bijvoorbeeld lopen van -32768 tot en met 32767. Elke Ada versie moet minstens het INTEGER datatype bevatten en bevat verder al of niet LONG_INTEGERs en SHORT_INTEGERs. Het gebruik van de voorgedefinieerde integer typen wordt echter niet aanbevolen, omdat hiermee een stuk machine-afhankelijkheid wordt geïntroduceerd (het waardebereik kan immers van machine tot machine verschillen), zodat de overdraagbaarheid van de programmatuur bemoeilijkt wordt.

Veel beter is het als de programmeur zelf expliciet een integer type declareert met een expliciet aangegeven waardebereik. De compiler zorgt vervolgens zelf voor een correcte binaire voorstellingswijze en dit bevordert de overdraagbaarheid van de programmatekst ('source code'). De declaratie maakt gebruik van het gereserveerde woord **range**, gevolgd door de onder- en bovengrens van het integer interval (de *'range constraint'*). Dit zijn voorbeelden van integer type definities:

```
type REGEL_TELLER is range     0..66;
type INDEX        is range    55..77;
type DIEPTE       is range -5000.. 0;
```

De twee puntjes zijn verplicht en ze staan voor waarden die liggen tussen de onder- en de bovengrens. Deze begrenzingen moeten statische waarden zijn; dat wil zeggen dat zij op het moment van compilatie van het programma bekend moeten zijn.

Bovenstaande structuur biedt ons opnieuw een mogelijkheid tot abstractie. Objecten van het type REGEL_TELLER hadden we ook als INTEGER kunnen declareren, maar dan zou ons model verder afstaan van de werkelijkheid. Negatieve aantallen regels komen nu eenmaal niet voor en als het gaat om regels op een pagina dan is een interval 0..66 een beter model van de realiteit. En dat niet alleen: Ada zorgt er nu voor dat we niet buiten dit waardebereik kunnen komen. Stel bijvoorbeeld dat er een object REGELS wordt gedeclareerd:

```
REGELS : REGEL_TELLER;
```

Als aan REGELS nu ooit een waarde kleiner dan 0 of groter dan 66 zou worden toegekend, dan zou CONSTRAINT_ERROR dit uitzonderingsgeval opvangen en zou worden aangegeven dat er een poging was gedaan het waardebereik van REGELS te doorbreken. (Zie hoofdstuk 17 voor de wijze van opvangen van uitzonderingen, of 'exception handling'.)

De onder- en bovengrens van een waardeninterval hoeven niet te bestaan uit getallen, ook ingewikkelder expressies zijn toegelaten. Stel bijvoorbeeld dat eerder een constante RIJEN (met waarde 24) en een constante KOLOMMEN (waarde 80) werd gedeclareerd, dan mag ook:

```
type AANTAL_ELEMENTEN is range 1..(RIJEN*KOLOMMEN);
```

Bij de compilatie is bekend dat AANTAL_ELEMENTEN waarden mag aannemen uit het interval 1 tot en met 1920.

Bij de integer typen behoren een aantal voorgedefinieerde operaties, zoals in tabel 8-1 weergegeven. Deze tabel geeft ook een samenvatting van de bijbehorende attributen. In appendix D worden deze attributen nader omschreven.

Reële datatypen. Een *reëel type* bepaalt een waardenverzameling, die bij benadering overeenkomt met de verzameling der reële getallen. Omdat er op elke machine maar een eindig aantal bits ter beschikking is, om gegevens voor te stellen, is het onmogelijk alle reële waarden exact voor te stellen. Zelfs de decimale waarde 0.1 (één tiende) kan in binaire vorm niet exact worden voorgesteld, omdat het een binaire repeterende breuk is. (De binaire voorstelling is $0.0001\underline{10011}..$, waarbij achter de punt oplopende machten van $\frac{1}{2}$ staan. De decimale waarde is dus $(\frac{1}{2})^4 + (\frac{1}{2})^5 + (\frac{1}{2})^8 + (\frac{1}{2})^9 + \ldots$ en het repeterende deel is onderstreept.) De uitwerking van de declaratie van een reëel type leidt tot een eindige verzameling door de machine voorstelbare waarden. Deze waarden zijn op de meeste computers machten van twee en kunnen exact worden voorgesteld. De werkelijke reële

Tabel 8-1: Overzicht van Integer Datatypen

Waardenverzameling	verzameling van opeenvolgende gehele getallen						
Structuur	range L..U	L en U zijn statische expressies met gehele uitkomsten en stellen de onder- en bovengrens voor					
Operatieverzameling	optellen	+	-				
	toekennen	:=					
	machtsverheffen	**					
	expliciete conversie						
	element van verzameling	in	not in				
	vermenigvuldigen	*	/	mod	rem		
	kwalificatie						
	relatie	=	/=	<	<=	>	>=
	unaire operatie	+	-	abs			
Attributen	ADDRESS PRED BASE SIZE FIRST SUCC IMAGE WIDTH LAST VAL POS VALUE						
Voorgedefinieerde typen	INTEGER NATURAL SHORT_INTEGER LONG_INTEGER POSITIVE						

getallen in wiskundige zin van het type *universele real*, waarvan er niet-aftelbaar oneindig veel zijn, stellen zij slechts bij benadering voor en met een bepaalde precisie, afhankelijk van de implementatie.

Ook hier zijn er weer voorgedefinieerde typen beschikbaar, zoals:

```
COEFFICIENT : FLOAT;
AFSTAND     : LONG_FLOAT;
WEERSTAND   : SHORT_FLOAT;
```

De precisie van deze voorgedefinieerde typen is machine-afhankelijk: de precisie op een machine met 32 bit woorden en een 24 bits mantisse in de voorstelling van reële waarden is ongeveer 7 decimalen; een 64 bits voorstelling heeft ongeveer 16 significante cijfers.

Ook hier getuigt het weer van een betere programmeerstijl als niet van de voorgedefinieerde typen gebruik wordt gemaakt, maar als men de toegelaten benaderingsfout expliciet definieert en vervolgens de machine zelf tijdens de compilatie de juiste voorstellingswijze laat kiezen. De programmatuur wordt hierdoor beter overdraagbaar en de factoren die bij de probleemstelling een rol spelen

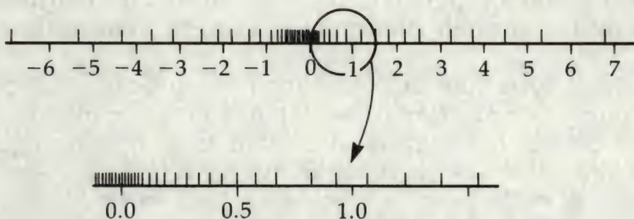
(in dit geval de numerieke precisie) worden zo op een expliciete wijze naar de oplossingsruimte afgebeeld. Bij de ontwikkeling van een besturingssysteem voor een ruimtevaartuig kan het bijvoorbeeld noodzakelijk zijn dat afstanden met een relatieve precisie van 15 significante cijfers wordt gemeten. Voor het meten van spanningen kan weer een absolute precisie tot op 0.001 Volt nauwkeurig nodig zijn. Het liefst zou men dergelijke precisie-eisen expliciet willen kunnen omschrijven en dit is in Ada dan ook mogelijk. We hoeven slechts een geschikte voorstellingswijze voor de gegevens te bepalen en de Ada compiler zal een machinevoorstelling kiezen met gelijke of grotere precisie, zodat aan de wiskundige eisen gesteld aan deze reële waarden in ieder geval voldaan is.

Om te beginnen kunnen we in Ada *floating-point typen* gebruiken om gegevens met een bepaalde relatieve nauwkeurigheid weer te geven. Zo'n floating-point definitie bestaat uit het gereserveerde woord **digits**, gevolgd door een statische expressie met een integer waarde, die het aantal significante decimale cijfers aangeeft. Bijvoorbeeld:

```
type PLANETAIRE_METING is digits 15;
type MASSA_BEPALING    is digits 7 range 0.0 .. 3.0;
```

Merk op hoe in het laatste voorbeeld een interval (range) specificatie met een precisie specificatie is gecombineerd. Ook hier moeten de intervalgrenzen statisch zijn, doch in dit geval dienen het reële waarden te zijn. Als we vervolgens een object van het type MASSA-_BEPALING declareren en als we proberen dit een waarde toe te kennen buiten het gespecificeerde interval, dan treedt de uitzonderingssituatie CONSTRAINT_ERROR in werking.

In figuur 8-3 wordt aangegeven hoe de declaratie van het floating-point type leidt tot een bepaalde weergave van getallen. Deze getallen zijn machten van twee, dat wil zeggen getallen die door een binaire computer exact kunnen worden voorgesteld. Hoe groter het gewenste aantal significante cijfers is, des te meer bits zijn er nodig voor de representatie. Verder geldt, dat de voorstelbare getallen, naarmate men verder van nul is, verder van elkaar af komen te liggen.



Figuur 8-3 Weergave van floating-point getallen

Declareren we

digits N;

dan moet de machinerepresentatie zodanig zijn, dat de mantisse, dat wil zeggen de voorgestelde breuk, minstens uit B bits bestaat, waarbij B het eerste gehele getal is met een waarde groter of gelijk:

N * log(10)/log(2) + 1

Voor 6 significante cijfers zijn bijvoorbeeld 21 bits nodig. Algemeen zijn voor elk cijfer $\log_2(10) = 3.321928095$ bits nodig. Als de machine de gewenste precisie niet aan kan (bijvoorbeeld **digits** 100), dan zal de compiler deze specificatie weigeren.

Floating-point getallen worden algemeen voorgesteld als:

$M * R^X$

waarbij M de *mantisse* voorstelt (meestal een getal tussen -1 en 1), R het grondtal of de *radix* en X de *exponent*. Als M een binaire fractie voorstelt van B bits, en het grondtal R is gelijk aan 2, dan zal X ook minimaal de waarde B moeten kunnen aannemen, om het gewenste aantal significante cijfers te kunnen bereiken. Indien de exponent een groter waardebereik heeft, dan wordt de getalsvoorstelling *veilig* genoemd.

Een berekening kan tot een resultaat leiden dat tussen twee voorstelbare floating-point waarden in ligt. De machine-implementatie zal dan een waarde kiezen binnen dit interval, zodat de gewenste precisie gegarandeerd blijft. Bijvoorbeeld:

```
MASSA : MASSA_BEPALING;
. .
MASSA := 2.314_587_936;
```

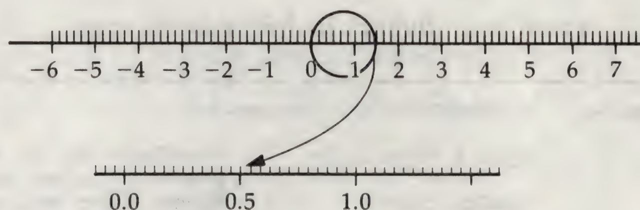
MASSA is een object met 7 significante cijfers; het getal 2,314_587_936 dat 10 significante cijfers heeft ligt dus tussen twee mogelijk waarden van MASSA in. Bij de waardetoekenning zal er nu voor gezorgd worden dat op zijn minst de precisie tot in 7 cijfers gehandhaafd blijft. De taalregels ondersteunen dus onze abstracte voorstellingswijze van MASSA.

Een numeriek type kan ook een absolute precisie hebben via een *fixed-point voorstelling*. Hierbij wordt het gereserveerde woord **delta** gebruikt om de vaste afstand tussen twee opeenvolgende waarden aan te geven. Het woord **delta** wordt gevolgd door een statische expressie met een reële waarde. Ook hier kan weer gelden dat de machine-implementatie leidt tot een kleinere stapgrootte, maar de gespecificeerde stapgrootte wordt in ieder geval gegarandeerd. Voorbeeld:

```
type STROOM_METING  is delta 0.025 range 0.0 .. 100.0;
type VOLTAGE        is delta 0.1   range -12.0 .. 24.0;
```

Tabel 8-2: Overzicht Reële Datatypen

Waardenverzameling	benadering tot de reële getallen	
Structuur	digits N range L..U (floating point)	specificatie van de relatieve precisie. N is het aantal significante cijfers en L en U zijn (desgewenst) onder- en bovengrens van het waardenbereik
	delta D range L..U (fixed point)	specificatie van absolute precisie. D is een reële waarde voor de stapgrootte. L en U zijn (verplichte) onder- en bovengrens van het waardenbereik
Operatieverzameling	optellen	+ -
	toekennen	:=
	machtsverheffen	**
	expliciete conversie	
	element van verzameling	in not in
	vermenigvuldiging	* /
	kwalificatie	
	relatie	= /= < <= > >=
	unaire operatie	+ - abs
attributen	*fixed-point;*	
	ADDRESS	LAST
	AFT	MACHINE_OVERFLOWS
	BASE	MACHINE_ROUNDS
	DELTA	MANTISSA
	FIRST	SIZE
	FORE	SAFE_LARGE
	LARGE	SAFE_SMALL
		SMALL
	floating-point;	
	ADDRESS	MACHINE_MANTISSA
	BASE	MACHINE_OVERFLOWS
	DIGITS	MACHINE_RADIX
	EMAX	MACHINE_ROUNDS
	EPSILON	MANTISSA
	FIRST	SAFE_EMAX
	LARGE	SAFE_LARGE
	LAST	SAFE_SMALL
	MACHINE_EMAX	SIZE
	MACHINE_EMIN	SMALL
Voorgedefinieerde typen	FLOAT LONG_FLOAT SHORT_FLOAT	



Figuur 8-4 Voorstelling van fixed-point getallen

In dit geval is het toevoegen van het gewenste interval verplicht.

Zoals figuur 8-4 aangeeft, liggen de fixed-point voorstellingen op onderling gelijke afstand. De tussenruimte is de 'fijnheid' van de voorstellingswijze. Elk fixed-point getal wordt door één van deze waarden voorgesteld en voldoet minimaal aan de gespecificeerde eigenschappen.

In tabel 8-2 worden de operaties en attributen van Ada's reële typen opgesomd. Merk op dat machtsverheffing alleen is gedefinieerd voor geheeltallige machten; via de voorgedefinieerde operator kan bijvoorbeeld 9**(0.5) dus niet berekend worden, maar in hoofdstuk 10 zullen we zelf een machtsverheffingsoperator voor reële machten ontwikkelen.

Er is een reden voor de vele attributen voor reële typen in Ada: in numerieke toepassingen moet men rekening houden met de eindige voorstellingswijze van reële getallen in de computer en de beperkingen die daaruit volgen. Met behulp van de beschikbare attributen kunnen numerieke benaderingsmethodes tot een tevoren vastgestelde precisie worden uitgevoerd.

We vermeldden al eerder dat het uitvoeren van operaties tussen objecten van verschillende typen niet is toegelaten. Het direct proberen op te tellen van een reële en een geheeltallige waarde doorbreekt onze abstractieniveaus en wordt al tijdens de compilatie als fout ontdekt. In plaats van impliciete conversie, zoals in veel andere programmeertalen (een vaak niet overdraagbare en daarom gevaarlijke operatie), moet typeconversie in Ada expliciet worden gespecificeerd:

```
INTEGER(-1.9)
                -- converteer naar de dichtsbijzijnde INTEGER waarde
MASSA_BEPALING(2)   -- converteer naar floating-point waarde
```

Expliciete conversie is toegelaten tussen alle numerieke typen en wordt uitgevoerd door voor de expressie tussen haakjes de naam van het gewenste numerieke type te plaatsen.

Enumeratietypen. Niet alle objecten uit onze probleemgebieden kunnen eenvoudig als numerieke waarden worden voorgesteld. Om een voorbeeld te geven: in een pak speelkaarten gaat het niet om getallen, maar om de waarden KLAVEREN, RUITEN, HARTEN en SCHOPPEN. Evenzo is de positie van een landingsgestel van een vliegtuig niet nul of één, maar OP of NEER. In een taal als FORTRAN kan men wel de objecten, die een pak speelkaarten of een landingsgestel voorstellen, definiëren, maar hun mogelijke waarden zijn steeds getallen. De gebruiker zal steeds zelf in gedachten de transformatié naar de realiteit moeten uitvoeren. Dit leidt niet tot erg leesbare programma's; en we zagen al in hoofdstuk 4 dat leesbaarheid van groot belang is voor de onderhoudbaarheid.

Een type geeft een verzameling waarden aan en het zou beter zijn als de programmeur de mogelijke waarden ook expliciet zou kunnen opsommen. Dit gebeurt nu in Ada via het *enumeratietype*, in de vorm van een geordende lijst waarden. Voorbeelden:

```
type PAK_KAARTEN            is (KLAVEREN,RUITEN,HARTEN,SCHOPPEN);
type POSITIE_LANDINGSGESTEL is (NEER,OP);
type MOTOR_RICHTING         is (UIT,VOORUIT,ACHTERUIT);
type HEX_CIJFER             is ('A','B','C','D','E','F');
```

Uit het laatste voorbeeld blijkt, dat lettertekens ook als enumeratiewaarden kunnen worden gebruikt.

Vervolgens kunnen objecten van de gecreëerde typen worden gedeclareerd:

```
SPEELKAART     : PAK_KAARTEN;
LANDINGSGESTEL : POSITIE_LANDINGSGESTEL;
HIJSKRAAN      : MOTOR_RICHTING;
. . .
SPEELKAART := PAK_KAARTEN'EERSTE    -- de waarde KLAVEREN
LANDINGSGESTEL := OP;
HIJSKRAAN := ACHTERUIT;
```

Alleen de opgesomde waarden kunnen aan de variabelen van elk enumeratietype worden toegekend; schending van ons abstractieniveau is opnieuw niet toegelaten. Merk op dat het hier om elementaire waarden gaat; hoe de onderliggende implementatie is (mogelijkerwijs via integers) is hier niet van belang. Ada beschouwt de enumeratielijst als een van links naar rechts geordende lijst.

In tabel 8-3 worden de operaties en attributen bij enumeratietypen opgesomd. Dit type kent geen operaties, zoals optellen of aftrekken, maar wel de ordeningsrelaties. Met behulp van de attributen kunnen ook de volgende waarden uit de enumeratielijst worden gehaald:

```
PAK_KAARTEN'SUCC(HARTEN)
                    -- SCHOPPEN is de opvolger van HARTEN
LANDINGSGESTEL'LAST    -- de waarde OP
```

Tabel 8-3: Overzicht Enumeratietypen

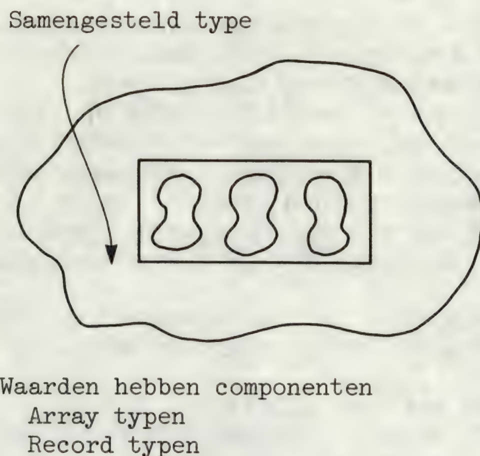
Waardenverzameling	een geordende verzameling verschillende waarden	
Structuur	$(E_0,E_1,...,E_n)$	E_i is een geordende enumeratiewaarde
Operatieverzameling	toekenning	:=
	element van verzameling	in not in
	kwalificatie	
	relatie	= /= < <= > >=
Attributen	ADDRESS	PRED
	BASE	SIZE
	FIRST	SUCC
	IMAGE	VAL
	LAST	VALUE
	POS	WIDTH
Voorgedefinieerde typen	BOOLEAN	
	CHARACTER	

Ook voor enumeratietypen geldt weer dat, als we een niet bestaande waarde proberen te genereren (bijvoorbeeld de voorganger van KLAVEREN), de situatie CONSTRAINT_ERROR ontstaat.

Ada heeft twee voorgedefinieerde enumeratietypen: BOOLEAN en CHARACTER. Zoals in appendix C wordt beschreven in het STANDARD pakket, heeft het type BOOLEAN twee waarden FALSE en TRUE (in die volgorde) en omvat het type CHARACTER de standaard ASCII (American Standard Code for Information Interchange) lettertekenset. De waarden van het type CHARACTER zijn lettertekens, weergegeven door een teken tussen twee accenten, zoals 'B'.

Samengestelde datatypen

Sommige objecten uit een probleemgebied, zoals VOLTAGE of STROOMSNELHEID, zijn niet uit verschillende delen samengesteld en kunnen dus als scalaire typen worden voorgesteld. Vaak bestaan objecten echter logischerwijs uit meer dan een component. Een WERKNEMER_RECORD kan bijvoorbeeld componenten bevatten als NAAM, AFDELINGSNUMMER, GEBOORTEDATUM en ADRES, terwijl bijvoorbeeld de MATRIX bestaat uit een tabel met rijen en kolommen van dezelfde componenten. Ada kent *samengestelde typen* (composite types) om dergelijke gegevens te beschrijven en figuur 8-5 illustreert dit. Samengestelde typen kunnen in twee categorieën worden onderverdeeld:




```
Waarden hebben componenten
  Array typen
  Record typen
```

Figuur 8-5 Samengestelde datatypen

- array typen
- record typen

Een *array* of *rij* is een genummerde verzameling van gelijksoortige objecten. Een *record* kan zijn samengesteld uit ongelijksoortige objecten.

Array typen. De naam van een array verwijst naar een verzameling componenten, die alle van hetzelfde type zijn. Een component van een array kan worden geselecteerd met behulp van één of meer indexwaarden. Het aantal indices heet de dimensie van het array. Een array met één index is ééndimensionaal, een array met twee indices is tweedimensionaal, enzovoort. Aan het aantal dimensies zijn in Ada geen beperkingen opgelegd.

Een *array type* wordt gespecificeerd door de intervallen van de indices aan te geven en het type van de componenten. Voorbbelden:

```
type SCHAAKBORD is array (1..8,1..8) of SCHAAK_STUK;
type LIJST is array (INTEGER range -100..10) of FLOAT;
type VECTOR is array(INTEGER range 1..MAXIMUM_INDEX)
            of FLOAT;
```

We kunnen nu objecten met deze typen declareren:

```
DONNERS_BORD       : SCHAAKBORD; -- een tweedimensionaal array
GESORTEERDE_LIJST  : LIJST;      -- een eendimensionaal array
TOESTANDSVECTOR    : VECTOR;     -- ook een eendimensionaal array
```

Door middel van een *geïndiceerde component* kan nu elk array-object direct worden benaderd. Bijvoorbeeld: DONNERS_BORD(1,5). In hoofdstuk 11 zullen wij hierop nader ingaan.

De hierboven gegeven voorbeelden laten nog een aantal bijzonderheden zien. In de declaratie van SCHAAKBORD is bijvoorbeeld het type van de indices niet aangegeven. Indien geheeltallige cijfers als intervalbegrenzingen worden gebruikt, wordt automatisch aangenomen dat de index van het type integer is. Bij het type VECTOR is als bovengrens een statische expressie met een geheeltallige waarde gebruikt.

Het is niet nodig altijd het voorgedefinieerde INTEGER type voor de indices te gebruiken. Andere mogelijkheden zijn:

```
type LANGE_INDEX is range 0..1000;
type LANG_ARRAY  is array (LANGE_INDEX) of FLOAT;
type KORT_ARRAY  is array (LANGE_INDEX range 10..49) of FLOAT;
```

Op deze manier definieert LANG_ARRAY een array type met 1001 elementen en KORT_ARRAY een array type met 40 elementen. Een kenmerk van goede programmeerstijl is het expliciet declareren van een datatype als LANGE_INDEX voor elk array, en vervolgens arrays te declareren, zoals hierboven is gedaan, in plaats van gebruik te maken van het voorgedefinieerde integer type. Declaratie van expliciete typen leidt tot een hogere graad van abstractie en misbruik van array indicering kan direct worden gesignaleerd.

De hierboven gegeven array declaraties zijn vrij eenvoudig; zoals in vrijwel iedere programmeertaal is het mogelijk voor array indices een integer type te kiezen. In Ada kan een array echter met behulp van ieder discreet type worden geïndiceerd, dus ook met enumeratietypen:

```
type DAG is (MAANDAG,DINSDAG,WOENSDAG,DONDERDAG,
             VRIJDAG,ZATERDAG,ZONDAG);
type UREN is delta 0.1 range 0.0 .. 24.0;
type UREN_RAPPORTAGE is array (DAG range MAANDAG ..
                                     VRIJDAG) of UREN;
type OVERWERK is array (DAG range ZATERDAG .. ZONDAG) of UREN;
type WEEK is array (DAG) of UREN;
URENBESTEDINGSOVERZICHT : UREN_RAPPORTAGE;
```

We kunnen nu refereren aan bijvoorbeeld URENBESTEDINGS-OVERZICHT(MAANDAG). De dagen gaven we aan door hun volledige naam, en in plaats van voor de uren het type FLOAT te kiezen, creëerden we een eigen type UREN, dat beter met de werkelijkheid overeenkomt.

Alle tot nu toe gedeclareerde array typen zijn begrensd, dat wil zeggen de grenzen zijn bekend als de type definitie wordt uitgevoerd. Alle op deze wijze gedeclareerde objecten van het type array hebben dus dezelfde omvang. In de programmeertaal Pascal is dit de enige mogelijkheid en dit is dan ook als een tekortkoming van Pascal

aangemerkt; vaak is er immers behoefte aan objecten van hetzelfde array type, maar met verschillende begrenzingen. In een systeem voor het verzenden en ontvangen van berichten bijvoorbeeld, behoeft niet ieder bericht dezelfde lengte te hebben. In Ada wordt dit probleem opgelost door declaratie van niet-begrensde array-typen toe te laten. De intervallen van de indices worden dan pas gespecificeerd op het moment dat een object van dit array-type wordt gedeclareerd. Deze begrenzingen hoeven niet statisch te zijn, maar kunnen tijdens de verwerking worden bepaald.

Niet begrensde arrays worden met behulp van een zogenaamde *boxnotatie*, bestaande uit de tekens <>, beschreven. Voorbeelden:

```
type INDEX is range 1..64;
type BIT_VECTOR is array (INDEX range <>) of BOOLEAN;
type MATRIX ia array (INDEX range <>,INDEX range <>) of FLOAT;
```

Ada vereist dat de begrenzingen van een array object bekend zijn op het moment dat het object gecreëerd wordt. Dan moeten de intervallen dus worden aangegeven:

```
    FILTER         : BIT_VECTOR(1..31);
    TRANSFORMATIE  : MATRIX(1..4,1..3);
    INVERSE        : MATRIX(1..N,1..N);
```

Op deze wijze kunnen de begrenzingen van array-typen via een soort parameter overdracht gespecificeerd worden. Merk op dat de waarde van N in het laatste voorbeeld het resultaat kan zijn van een tijdens de verwerking gemaakte berekening. De begrenzingen van een array-index kunnen, zoals in bovenstaande voorbeelden, via een declaratie worden gespecificeerd; is echter een array-object een formele parameter voor een subprogramma, dan worden de begrenzingen van de na de aanroep actuele parameter aangenomen (zie hoofdstuk 10).

Het pakket STANDARD bevat ook een array-type STRING genaamd. Dit is een ééndimensionaal array met componenten van het type CHARACTER. Voorbeelden van declaraties van objecten van dit type:

```
    NAAM   : STRING(1..18);
    PROMPT : constant STRING := "VOER DE GEGEVENS NU IN: ";
```

In dit laatste geval wordt de lengte van de string afgeleid uit zijn beginwaarde, in dit geval geldt: PROMPT'LENGTH = 24.

Objecten van het type STRING (maar ook andere ééndimensionale arrays) kunnen geconcateneerd worden, dat wil zeggen een tweede string kan achter een eerste worden 'geplakt':

```
    NAAM := "BRANDSTOF TANK" & " NUMMER 7";
```

Resultaat is een string ter lengte 23.

Tabel 8-4: Overzicht van Array-Datatypen

Waardenverzameling	een geïndexeerde verzameling van gelijksoortige typen	
Structuur	array (index{,index}) of component	
	(onbegrensd array)	index{,index} is een rij van onbegrensde discrete typen; component geeft het type aan van de waarden in het array.
	array index_begrenzing of component	
	(begrensd array)	indexbegrenzing is een lijst van discrete type; component geeft het type aan van de waarden in het array.
Operaties	optelling (ééndimensionaal array)	&
	aggregatie	
	waardetoekenning	:=
	expliciete conversie	
	indexering	
	logisch (boole's componenten)	and or xor not
	element van verzameling	in not in
	kwalificatie	
	relatie	= /=
	relatie (discrete componenten)	< <= > >=
	unair (boole'se componenten)	not
Attributen	ADDRESS	LAST(J)
	BASE	LENGTH
	FIRST	LENGTH(J)
	FIRST(J)	RANGE
		RANGE(J)
	LAST	SIZE
Voorgedefinieerde typen	STRING	

In tabel 8.4 worden de eigenschappen van arrays in Ada nog eens samengevat. In hoofdstuk 11 worden de mogelijke array-operaties nog nader bestudeerd.

Record datatypen. Met array-typen kunnen we goed uit de voeten als alle elementen hetzelfde type hebben. Vaak bestaat een object echter uit een groep ongelijksoortige componenten. In dat geval kan het record type van pas komen. Een *record type* wordt gedeclareerd met behulp van de gereserveerde woorden **record .. end record**. Deze woorden sluiten de recordcomponenten in.

De syntaxis van de recordcomponenten komt overeen met die van andere objectdeclaraties. Voorbeelden:

```
type DAG_VAN_HET_JAAR is
  record
    DAG   : INTEGER range 1 .. 31;
    MAAND : MAAND_NAAM;
    JAAR  : NATURAL;
  end record;

type CPU_VLAG is
  record
    CARRY     : BOOLEAN;
    INTERRUPT : BOOLEAN;
    NEGATIEF  : BOOLEAN;
    NUL       : BOOLEAN;
  end record;

type CPU_STATUS is
  record
    PRIORITEIT : INTEGER range 0 .. 7;
    VLAG       : CPU_VLAG;
  end record;

DATUM : DAG_VAN_HET_JAAR;
PSW   : CPU_STATUS;
```

Merk op dat er in het geval van PSW sprake is van een geneste recordstructuur. Objecten die componenten zijn van een record kunnen via een component selectienotatie worden aangeroepen (zie ook hoofdstuk 11). In de volgende voorbeelden gebeurt dit:

```
DATUM.JAAR
PSW.PRIORITEIT
PSW.VLAG.NUL
```

In tabel 8-5 zijn de bijzonderheden van het record datatype nog eens samengevat. Operaties zijn nauwelijks mogelijk op record (of array) typen. Records zijn dan ook voornamelijk bedoeld om structuur aan samengestelde gegevens te geven. Vanzelfsprekend kunnen op de componenten van arrays en records wel alle operaties worden toegepast, behorend bij het type van die componenten.

In de tot nu toe gegeven voorbeelden waren de recordcomponenten betrekkelijk onafhankelijk van elkaar. Zo hangen PSW.PRIORITEIT en PSW.VLAG.NUL bijvoorbeeld op geen enkele wijze van elkaar af. In bepaalde gevallen kan echter ofwel de recordstructuur, ofwel de waarde van een recordcomponent afhangen van de waarde van een andere component. Een record met vliegtuiggegevens kan bijvoorbeeld verschillende componenten hebben, afhankelijk van het type vliegtuig. Een voorbeeld van de onderlinge afhankelijkheid van waarden is het volgende: een transformatiematrix in een grafisch pakket moet vierkant zijn en we moeten er dan dus voor zorgen dat geen rechthoekige matrix kan worden gebruikt. Om dit mogelijk te maken

Tabel 8-5: Overzicht van Record Datatypen

Waardenverzameling	een verzameling van (mogelijk) verschillende componenten	
Structuur	record componentenlijst end record	de componentenlijst benoemt de samenstellende delen van het record
Operatieverzameling	toekenning	:=
	aggregatie	
	expliciete conversie	
	element van verzameling	in not in
	kwalificatie	
	relatie	= /=
	selectie	
Attributen	*record type*	
	ADDRESS	
	BASE	
	CONSTRAINT	
	SIZE	
	recordcomponent	
	FIRST_BIT	
	LAST_BIT	
	POSITION	
Voorgedefinieerde typen	geen	

wordt een zogenaamde *record discriminant* gebruikt, dat wil zeggen een grootheid die het mogelijk maakt verschijningsvormen van hetzelfde record van elkaar te onderscheiden. In dit laatste geval wordt de discriminant vermeld in de typedefinitie en kan binnen het record gebruikt worden als een indexbegrenzing, als een default waarde voor een component, of als een waarde voor een andere discriminant. De discriminant mag geen onderdeel zijn van een grotere expressie. Enige voorbeelden om dit te verduidelijken:

```
type GEWOON_ARRAY is array(INTEGER range <>,
                           INTEGER range <>) of FLOAT;
```

Objecten zouden nu als volgt kunnen worden gedeclareerd:

```
EERSTE_MATRIX : GEWOON_ARRAY(1 .. 4, 1 .. 5);
```

Niets dwingt ons er dus toe om een vierkante matrix te declareren. Op de volgende manier is dit wel het geval:

```
type VIERKANT(ZIJDE : INTEGER := 4) is
  record
    MATRIX : GEWOON_ARRAY(1 .. ZIJDE, 1 .. ZIJDE);
  end record;
```

De waarde 4 is nu de default waarde (dat wil zeggen de waarde die automatisch wordt aangenomen als niets wordt vermeld). In het volgende voorbeeld wordt de discriminant LENGTE binnen het record opnieuw als discriminant gebruikt:

```
type TWEE_VIERKANTEN(LENGTE : INTEGER) is
  record
    EERSTE : VIERKANT(LENGTE);
    TWEEDE : VIERKANT(LENGTE);
  end record
. . .
TRANSFORM_3D : VIERKANT;              -- bij default begrensd
TRANSFORM_2D : VIERKANT(3);           -- 3 bij 3
TRANSFORM    : VIERKANT(ZIJDE => 3); -- eveneens begrensd
```

Binnen een record mag in Ada geen array type zonder dimensies voorkomen, vandaar de dimensies bij de declaratie van MATRIX. Elementen van de matrix kunnen nu als volgt worden aangeroepen:

```
TRANSFORM_3D.MATRIX(2,3)
```

In bovenstaande aanroep wordt zowel selectie via de puntnotatie, als indexering toegepast. Het record TRANSFORM_3D wordt *niet begrensd* genoemd, omdat bij de declaratie geen waarde van de discriminant werd opgegeven. In de volgende twee gevallen gebeurt dit wel. In het eerste geval direct en in het tweede geval via een benoemde parameter.

Discriminanten kunnen niet aleen op de hierboven beschreven wijze worden gebruikt om afhankelijkheden tussen waarden aan te geven; er kunnen ook zogenaamde *variante records* mee worden gedefinieerd. Een record voor een vliegtuig in een of ander militair beveiligingssysteem kan er bijvoorbeeld zo uitzien:

```
type VLIEGTUIG_ID is (BURGER,MILITAIR,VIJAND,ONBEKEND);
type BEDREIGING   is (LAAG,GEMATIGD,HOOG);
type VLIEGTUIG_RECORD(SOORT : VLIEGTUIG_ID :=ONBEKEND) is
  record
    VLIEGSNELHEID  : SNELHEID;
    VLIEGRICHTING  : RICHTING;
    BREEDTE        : COORDINAAT;
    LENGTE         : COORDINAAT;
```

```
    case SOORT is
      when BURGER => null;
      when MILITAIR =>
        CLASSIFICATIE : MILITAIR_TYPE;
        OORSPRONG     : LAND;
      when VIJAND|ONBEKEND =>
        BEDREIGING : BEDREIGINGSNIVEAU;
    end case;
  end record;
VLIEGTUIG : VLIEGTUIG_RECORD;
```

Hoewel we dit niet aangaven, moeten alle gebruikte datatypen, zoals bijvoorbeeld COORDINAAT tevoren gedefinieerd zijn.

Het record in dit laatste voorbeeld heeft vier componenten, die bij elk SOORT van VLIEGTUIG voorkomen, namelijk: VLIEGSNELHEID, VLIEGRICHTING, BREEDTE, en LENGTE. De **case** instructie specificeert het variabele gedeelte en moet altijd als laatste component worden opgenomen. Na elke **when** staat een van SOORT afhankelijke component van het record. Als SOORT gelijk is aan BURGER, dan geeft **null** aan dat er geen verdere componenten zijn. Is SOORT daarentegen MILITAIR, dan zijn er nog twee componenten, namelijk CLASSIFICATIE en OORSPRONG. Als de structuur dezelfde is voor twee verschillende waarden van de discriminant, dan worden de waarden door een rechtopstaand streepje gescheiden (lees: 'als VIJAND of ONBEKEND dan').

Een mogelijkheid is in dit voorbeeld niet aangegeven: met behulp van het gereserveerde woord **others** kan worden gespecificeerd, welke componenten moeten worden toegevoegd indien de discriminant geen der opgesomde waarden bezit.

Het object VLIEGTUIG dat tenslotte werd gedeclareerd heeft geen discriminant meegekregen en in dat geval kan SOORT tegelijk met de overige componenten via een record-waardetoekenning een waarde krijgen. In het volgende voorbeeld wordt daartoe een *geaggregeerde waarde* gebruikt (zie ook hoofdstuk 11):

```
VLIEGTUIG := (SOORT          => MILITAIR,
              VLIEGSNELHEID  => 150.0,
              VLIEGRICHTING  =>  97.3,
              BREEDTE        => 147.6,
              LENGTE         =>  27.1,
              CLASSIFICATIE  => TRANSPORT,
              LAND           => FRANKRIJK);
```

Eerst moet de waarde van de discriminant worden opgegeven en vervolgens de waarde van de componenten. Ook hier zijn de parameters benoemd om de duidelijkheid te bevorderen.

Typen kunnen ook begrensd worden gedeclareerd:

```
BURGER_VLIEGTUIG   : VLIEGTUIG_RECORD(SOORT => BURGER);
ONBEKEND_VLIEGTUIG : VLIEGTUIG_RECORD(ONBEKEND);
```

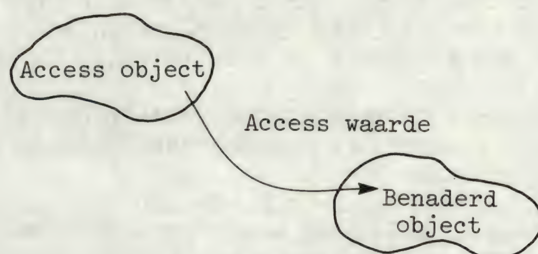
We kunnen nu werken met bijvoorbeeld BURGER_VLIEGTUIG-.VLIEGRICHTING en in het tweede geval met ONBEKEND_VLIEGTUIG.BEDREIGING (maar niet CLASSIFICATIE of OORSPRONG). Bij dit soort begrensde record typen mag SOORT niet worden veranderd.

Access datatypen

Ada is zodanig ontworpen, dat de semantiek (dat wil zeggen de aan de programmatekst te hechten betekenis) zoveel mogelijk statisch is en dus tijdens de verwerking niet verandert. Dit vergroot de betrouwbaarheid, omdat de logische analyse van het programma voor het grootste deel al tijdens de compilatie kan worden uitgevoerd. In Ada is het type van de meeste objecten daarom statisch, terwijl aantal, bereik en bereikbaarheid van grootheden voordat de verwerking begint al bekend zijn. Helaas moeten we concluderen dat te abstraheren objecten uit de wereld om ons heen vaak niet zo netjes statisch zijn. Dat wil zeggen [2]:

- Objecten moeten vaak op een niet van tevoren te voorspellen manier worden gecreëerd of verwijderd (denk aan buffers in een systeem voor uitwisselen van berichten).
- Meer dan één naam kan verwijzen naar hetzelfde object (een onderdeel uit een grafische weergave kan bijvoorbeeld onder meerdere namen bekend staan).
- De onderlinge samenhang tussen objecten kan met de tijd veranderen (bijvoorbeeld gegevens in een bevolkingsregister).

Met behulp van het *access type* kunnen dergelijke dynamische situaties in Ada worden opgevangen. Deze typen worden access of toegangstypen genoemd, omdat objecten van dit type toegang verschaffen tot andere objecten, zie figuur 8-6. Door een andere waarde te geven aan het toegangsobject kan ook een ander object worden



Figuur 8-6 Access datatypen

bereikt; en op deze manier kunnen ook nieuwe objecten worden gecreëerd, zoals we later in dit hoofdstuk zullen zien. Hierin verschillen objecten van het access type van andere typen, die via een vaste naam gecreëerd zijn en die statisch van aard zijn.

Hoe ziet een access-typedeclaratie eruit? Gebruikt wordt het gereserveerde woord **access** gevolgd door de naam van het type van de via het access te bereiken objecten. Een voorbeeld:

```
type BUFFER is
  record
    BERICHT     : STRING(1 .. 10);
    PRIORITEIT  : INTEGER range 1 .. 100;
  end record;
type BUFFER_POINTER is access BUFFER;
MIJN_PAKKET,UW_PAKKET,HUN_PAKKET : BUFFER_POINTER;
```

BUFFER_POINTER is het type dat toegang verschaft en BUFFER is het type van het object, waartoe toegang wordt gegeven. Om te beginnen hebben MIJN_PAKKET, UW_PAKKET en HUN_PAKKET alle de waarde **null**, want zij verwijzen nog naar niets. Let wel, dit is het *enige* geval, waarin Ada automatisch een default waarde, dat wil zeggen een waarde bij gebrek aan beter, toekent. Alle andere objecten hebben initieel géén waarde en zijn dus bijvoorbeeld ook niet automatisch gelijk aan nul, tenzij die waarde expliciet wordt toegekend.

Objecten van het type BUFFER kunnen nu worden gecreëerd met behulp van het gereserveerde woord **new**. Dit gaat als volgt:

```
MIJN_PAKKET := new BUFFER;
UW_PAKKET   := new BUFFER'(BOODSCHAP => "**********",
                           PRIORITEIT => 1);
HUN_PAKKET  := new BUFFER'("----------",10);
```

Deze instructies creëren elk een nieuw object van het type BUFFER, en door de **new** instructie ontvangen MIJN_PAKKET, UW_PAKKET en HUN_PAKKET een verwijzing naar deze objecten, zodat zij ook benaderd kunnen worden. In de laatste twee gevallen werd een zogenaamde geaggregeerde waarde (de constructie tussen haakjes) meegegeven om de BUFFER expliciet te initialiseren. (Zie ook hoofdstuk 11.)

Objecten kunnen niet alleen worden gecreëerd, maar ook weer worden vernietigd via access typen. Dat kan door alle verwijzingen naar het object te verwijderen:

```
MIJN_PAKKET :=new BUFFER;  -- een nieuw object van type BUFFER
MIJN_PAKKET :=null; -- het object kan niet meer worden benaderd
```

Deze dynamisch gecreëerde objecten worden trouwens automatisch vernietigd, zodra het bereik waarbinnen zij betekenis hebben wordt verlaten:

```
declare
  POINTER : BUFFER_POINTER;
begin
  . . .
  POINTER := new BUFFER;  -- creëer BUFFER object
  . . .
end;         -- BUFFER object wordt weer vernietigd
```

In beide gevallen kan de geheugenruimte, die door de BUFFER objecten werd ingenomen via een garbage collectie mechanisme weer worden vrijgegeven. (Garbage collection betekent letterlijk 'vuilnis ophalen' en dat is precies wat zo'n mechanisme doet: zoeken naar gebruikte, maar niet meer bezette geheugenruimte en deze weer vrijgeven.)

Zodra een object van het access type is gecreëerd kan gewerkt worden met zowel het verwijzende object als met het object waarnaar verwezen wordt. Dit laatste gaat met de al eerder gedemonstreerde puntnotatie:

```
MIJN_PAKKET.all             -- verwijst naar het hele object
MIJN_PAKKET.PRIORITEIT      -- verwijst naar de component PRIORITEIT
MIJN_PAKKET.BERICHT(1)      -- verwijst naar het eerste teken in BERICHT
```

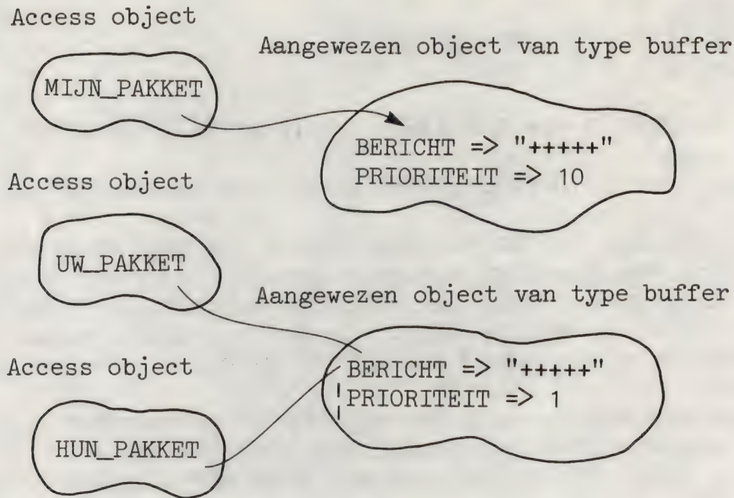
Het is van belang steeds een goed onderscheid te maken tussen de verwijzing en het object, waarnaar verwezen wordt. Het volgende voorbeeld illustreert dit:

```
MIJN_PAKKET := new BUFFER'(BERICHT => "+++",PRIORITEIT => 1);
                                  -- creëer BUFFER object
UW_PAKKET   := new BUFFER'(BERICHT => "+++",PRIORITEIT => 10);
                                  -- een tweede BUFFER object
HUN_PAKKET  := UW_PAKKET;  -- een verwijzing naar hetzelfde object
```

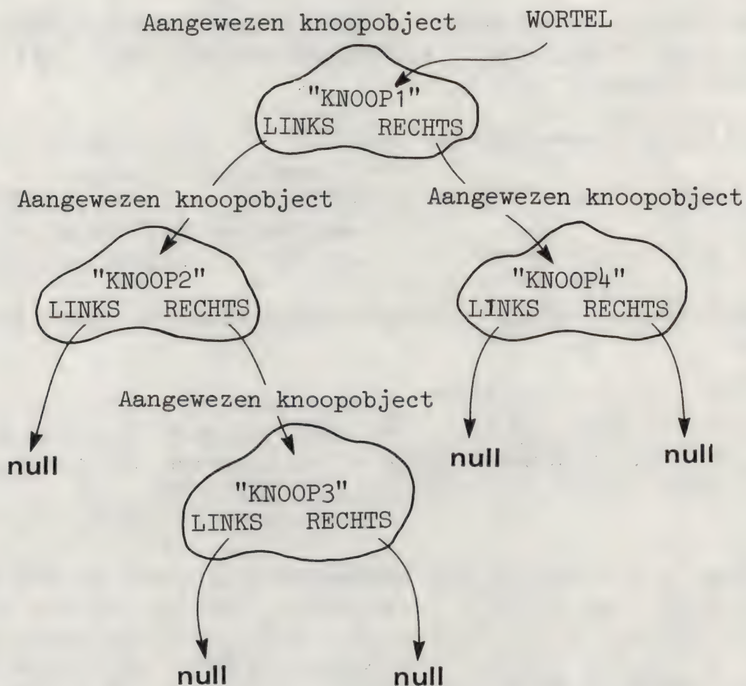
De onderstaande relaties (grafisch weergegeven in figuur 8-7) zijn alle waar:

```
MIJN_PAKKET /= UW_PAKKET   -- verwijzen naar verschillende objecten
UW_PAKKET = HUN_PAKKET     -- verwijzen naar hetzelfde object
UW_PAKKET.all /= MIJN_PAKKET.all  -- objecten zijn niet gelijk
MIJN_PAKKET.BERICHT = UW_PAKKET.BERICHT
                                  -- componenten hebben dezelfde waarden
```

We hebben nu aangegeven hoe access typen kunnen worden gebruikt om dynamisch objecten te creëren en te vernietigen en hoe verschillende namen naar hetzelfde object kunnen verwijzen. Access typen kunnen ook worden gebruikt om relaties tussen objecten aan te geven, met name als die in de tijd veranderen. Denk aan datastructuren, zoals verbonden lijsten ('linked lists') of binaire bomen. Een binaire boom, zoals in hoofdstuk 7 beschreven, zou bijvoorbeeld als volgt kunnen worden gemaakt:



Figuur 8-7 Samenhang tussen access waarden en objecten



Figuur 8-8 Aangeven van de samenhang van objecten via verwijzingen

```
type KNOOP;
type TAK is access KNOOP;
type KNOOP is
  record
    LINKS  : TAK;
    RECHTS : TAK;
    WAARDE : STRING(1 .. 5);
  end record;
WORTEL     : TAK;
TEMP_KNOOP : TAK;
POINTER    : TAK;
```

De eerste instructie 'type KNOOP;' ziet er wat vreemd uit; dit heet een *onvolledige typedeclaratie* en deze is noodzakelijk in het geval van recursieve of wederzijds van elkaar afhankelijke access-typen. Deze declaratie komt enigszins overeen met de later in dit

Tabel 8-6: Overzicht van Access Datatypen

Waardenverzameling	toegangswaarden tot aangewezen objecten	
Structuur	access subtype indicatie	de subtype indicatie geeft het type aan van het object waarnaar verwezen wordt
Operatieverzameling	allocatie	
	waardetoekenning	:=
	expliciete conversie	
	indexering (verwijzing naar array-objecten)	
	element van verzameling	in not in
	selectie (verwijzing naar records)	
	kwalificatie	
	relatie	= /=
Attributen	*access type* ADDRESS BASE SIZE STORAGE_SIZE	
	verwijzing naar array-objecten FIRST FIRST(J) LAST LAST(J) LENGTH LENGTH(J) RANGE RANGE(J)	*verwijzing naar taken* CALLABLE TERMINATED
Voorgedefinieerde typen	geen	

hoofdstuk te behandelen *private* en *limited private* typen: alleen het type wordt gedeclareerd, maar alle verdere informatie over waarden en operaties wordt weggelaten. Deze type declaraties moeten eerst worden uitgewerkt, voordat objecten van dit type kunnen worden gedeclareerd.

Met behulp van access objecten kan nu een binaire boom worden opgebouwd (zie figuur 8-8). Het volgende stukje Ada-programma bouwt deze boom op:

```
WORTEL              :=new KNOOP;       -- maak eerste knoop
WORTEL.WAARDE       :="KNOOP1";        -- geef knoop een waarde
TEMP_KNOOP          :=new KNOOP;       -- maak volgende knoop
TEMP_KNOOP.WAARDE   :="KNOOP2";        -- geef waarde
WORTEL.LINKS        :=TEMP_KNOOP;      -- verbind de knopen
TEMP_KNOOP          :=new NODE;        -- maak derde knoop
TEMP_KNOOP.WAARDE   :="KNOOP3";        -- geef waarde
POINTER             :=WORTEL.LINKS;    -- verwijs naar KNOOP2
POINTER.RECHTS      :=TEMP_KNOOP;
                                  -- verbind KNOOP2 en KNOOP3
TEMP_KNOOP          :=new KNOOP;
                                  -- maak een vierde knoop object
TEMP_KNOOP.WAARDE   :="KNOOP4";        -- geef het een waarde
WORTEL.RECHTS       :=TEMP_NODE;
                                  -- verbind WORTEL met KNOOP4
```

De boomstructuur kan nu eenvoudig gewijzigd worden. Bijvoorbeeld:

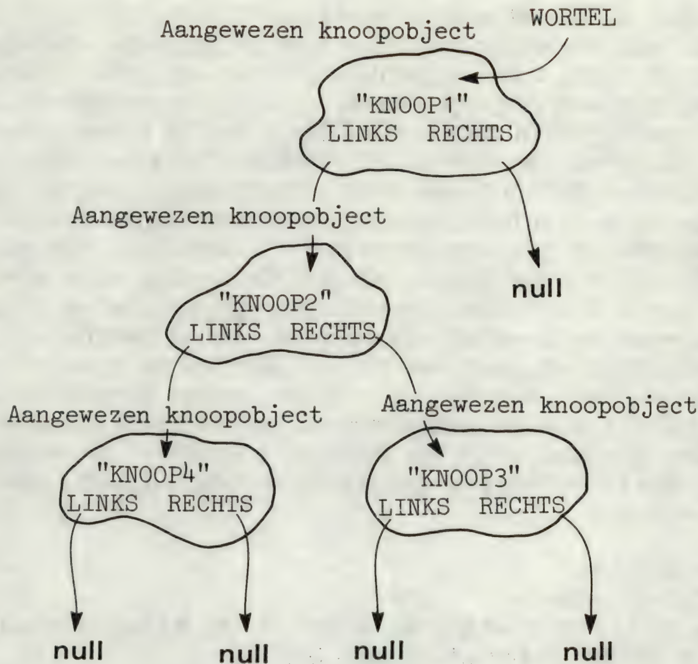
```
POINTER        := WORTEL.LINKS;   -- wijs naar KNOOP2
POINTER.LINKS  := WORTEL.RECHTS;  -- verbind met KNOOP4
WORTEL.RECHTS  := null;           -- verbreek oude verbinding
```

De nieuwe boom ziet er nu uit, zoals in figuur 8-9.

Private datatypen

De tot nu toe besproken datatypen (scalair, samengesteld en access) kunnen worden gebruikt voor het beschrijven van elementaire objecten in programma's. Een aantal mogelijke operaties op deze objecten zijn al door de taal gedefinieerd. Toch zijn deze datatypen niet voldoende om abstracte voorwerpen uit ieder probleemgebied te creëren. Zo kent Ada bijvoorbeeld geen COMPLEX_GETAL type en geen STACK type, maar Ada biedt wel de mogelijkheid onze eigen typen te definiëren met behulp van de *private type declaratie*. Deze private of 'privé' typen maken het mogelijk, informatie beperkt toegankelijk te maken ('information hiding'), zodat de wijze waarop operaties op een lager abstractieniveau zijn uitgewerkt voor de gebruiker niet zichtbaar is.

Een type bepaalt een waardenverzameling en een verzameling van toegelaten bewerkingen. Bij objecten van het type **private** zijn



Figuur 8-9 Gewijzigde samenhang tussen de objecten

zowel structuur als de waarden voor de gebruiker onzichtbaar; hij heeft alleen toegang tot de expliciet benoemde operaties. Private datatypen worden gedeclareerd in een pakketspecificatie door het type te laten volgen door *private* of *limited private*. Vervolgens worden de namen gespecificeerd van de subprogramma's, die de operaties definiëren, die op de objecten van het type private zijn toegelaten. Een voorbeeld:

```
package TOEGANGSCODE is
   type WAARDE is limited private;
   function IS_TOEGELATEN(CODE : in WAARDE) return BOOLEAN;
   procedure ZET(CODE : out WAARDE; BEVOEGDHEID : in NATURAL);
private
   type WAARDE is new STRING(1 .. 40);
end TOEGANGSCODE;
```

Nu kunnen objecten van het type WAARDE door een gebruiker van het pakket worden gedeclareerd:

```
GEBRUIKERS_BEVOEGDHEID : TOEGANGSCODE.WAARDE
```

Het pakket definieert een private type TOEGANGSCODE.WAARDE genaamd. De enig mogelijke operaties van dit type zijn IS_TOEGELATEN en ZET. Zelfs tests op gelijkheid van twee grootheden, of ongelijkheid, zijn niet mogelijk. Merk op dat de grootheid WAARDE intern wordt voorgesteld door een STRING van 40 tekens, maar dit betekent in dit geval niet dat het mogelijk is stringoperaties toe te passen op objecten van het type TOEGANGSCODE.WAARDE. (Het private gedeelte is in Ada ingevoerd, om afzonderlijke compilatie van pakketspecificatie en pakketbody mogelijk te maken.) Door een private of limited private type wordt het datatype als het ware ingekapseld en wordt de abstracte voorstellingswijze, onafhankelijk van de onderliggende implementatie bevorderd, omdat immers alleen de expliciet gedeclareerde operaties kunnen worden toegepast.

In het geval van limited private typen moeten inderdaad alle operaties expliciet worden gedeclareerd. In het geval van gewone private typen zijn drie operaties impliciet toegelaten: waardetoekenning (assignment) en tests op gelijkheid en ongelijkheid. Hier volgt weer een voorbeeld:

```
package RANDOM is
  type GETAL is private;
  procedure VUL(BEGINWAARDE : in INTEGER; WAARDE in out GETAL);
  function UNIFORM_RANDOM return GETAL;
private
  type GETAL is
    record
      EERSTE_WAARDE : INTEGER; WAARDE : FLOAT;
    end record;
end RANDOM;
GEBEURTENIS : RANDOM_GETAL;
```

Door alle grootheden betekenisvolle namen te geven is nu meteen duidelijk wat de aard is van de grootheid GEBEURTENIS. De toegelaten operaties zijn: het VULlen van de beginwaarde voor de randomgenerator en het aanroepen van deze generator voor een trekking uit de uniforme verdeling. In dit geval zijn verder toekenningen van waarden van RANDOM.GETAL-objecten aan objecten van ditzelfde type toegelaten en kunnen deze waarden op gelijkheid en ongelijkheid getest worden. Niet toegelaten zijn echter FLOAT operaties zoals vermenigvuldiging of deling; pogingen daartoe zullen tot een foutmelding leiden.

We hebben hiermee laten zien dat private typen een zeer belangrijk mechanisme vormen voor het creëren van abstracte datatypen en het verborgen houden van hun uitwerking op een lager niveau. In hoofdstuk 13 wordt de vorm van private typen in Ada verder beschreven en wordt ingegaan op het verband met pakketdeclaraties. Volledigheidshalve geeft tabel 8-7 nog eens een samenvatting van de eigenschappen van private datatypen.

Tabel 8-7: Overzicht van Private Datatypen

Waardenverzameling	voor de gebruiker niet toegankelijk
Structuur	voor de gebruiker niet zichtbaar
Operatieverzameling	expliciete conversie element van een verzameling **in** **not in** kwalificatie Voor **limited private** zijn alleen de operaties in de pakket-specificatie van toepassing; voor **private** ook nog waarde-toekenning en tests op gelijkheid en ongelijkheid.
Attributen	*Private typen* ADDRESS BASE SIZE *Private typen met discriminanten* CONSTRAINED
Voorgedefinieerde typen	geen

Subtypen en afgeleide typen

Onderverdeling van eigenschappen is één van de redenen waarom in Ada voor expliciete datatypedefinities werd gekozen. Soms moeten data echter nog verder in onderdelen worden opgesplitst dan mogelijk is met de tot nu toe behandelde typen. We kunnen bijvoorbeeld een type MAAND_NAAM hebben en alleen over de ZOMER maanden willen spreken, of we willen bijvoorbeeld verschil kunnen maken tussen een KALENDER_JAAR en een BELASTING_JAAR. In deze gevallen komen Ada's subtypen en afgeleide typen van pas.

Subtypen. Een *subtype* definieert geen nieuw type, maar maakt het mogelijk een nieuwe naam te geven aan een al bestaand datatype, in veel gevallen onder toevoeging van een bepaalde randvoorwaarde of beperking. Deze beperking hoeft, met uitzondering van de floating point of fixed point precisiebeperking, niet statisch te zijn, maar mag tijdens de verwerking worden bepaald. We kunnen bijvoorbeeld subtypen declareren zoals:

```
type MAAND_NAAM is (JANUARI,FEBRUARI,MAART,APRIL,
                    MEI,JUNI,JULI,AUGUSTUS,SEPTEMBER,
                    OKTOBER,NOVEMBER,DECEMBER);
subtype ZOMER      is MAAND_NAAM range JUNI .. AUGUSTUS;
DEZE_MAAND   : MAAND_NAAM;
VAKANTIETIJD : ZOMER;
```

ZOMER werd gedeclareerd als subtype van MAAND_NAAM. Dit laatste type wordt de *basis* voor het subtype genoemd.

Omdat het niet gaat om verschillende typen, kunnen zij in operaties rustig door elkaar gebruikt worden:

```
DEZE_MAAND   := VAKANTIETIJD;
VAKANTIETIJD := ZOMER'SUCC(DEZE_MAAND);
```

Als echter in de laatste waardetoekenning, DEZE_MAAND niet gelijk is aan MEI, JUNI of JULI, dan treedt de uitzonderingssituatie CONSTRAINT_ERROR in werking, omdat de waarde buiten de aangegeven begrenzingen valt.

Van subtypen kan handig gebruik worden gemaakt als we bij onze proboeemformulering deelverzamelingen van een bepaald basistype nodig hebben. Ook precisiebeperkingen, indexbeperkingen of discriminantbeperkingen kunnen op deze wijze worden ingevoerd. Bekijk de volgen de typen declaraties eens:

```
type NIET_NEGATIEF is range 0 .. INTEGER'LAST;
type GEWICHT       is delta 0.01 range 2000.0 .. 3000.0;
type VECTOR        is array (NATURAL range <>) of FLOAT;
type VOELER_KLASSE is (VOCHTIGHEID,DRUK,TEMPERATUUR);
type VOELER(SOORT : VOELER_KLASSE) is
  record
    PLAATS : STRING(1 .. 20);
    case SOORT is
      when VOCHTIGHEID  => WAARDE : INTEGER range 0 .. 100;
      when DRUK         => WAARDE : FLOAT;
      when TEMPERATUUR  => WAARDE : INTEGER range -50 .. 250;
    end case;
  end record;
```

We kunnen bijvoorbeeld de volgende subtypen declareren:

```
subtype INDEX          is NIET_NEGATIEF range 0 .. 100;
                                      -- range beperking
subtype BRUTO_GEWICHT  is GEWICHT delta 10.0;
                                      -- precisiebeperking
subtype VECTOR_3D      is VECTOR(1 .. 3);
                                      -- indexbeperking
subtype WARMTE_VOELER  is VOELER(SOORT => TEMPERATUUR);
                                      -- discriminantbeperking
```

Zolang niet gezondigd wordt tegen de beperkingen, die aan het subtype zijn opgelegd, kunnen objecten van basistype en subtype vrijelijk door elkaar worden gebruikt. We kunnen bijvoorbeeld objecten van het type INDEX en van het type NIET_NEGATIEF rustig bij elkaar optellen, maar zodra daar voor INDEX een waarde groter dan 10 uit zou volgen, treedt weer de uitzonderingssituatie CONSTRAINT_ERROR in werking. Hoewel dit uit bovenstaand

voorbeeld niet blijkt, behoeft alleen de precisiebeperking een statische expressie te zijn.

Natuurlijk moet iedere subtypedeclaratie in ieder geval voldoen aan de beperkingen die aan het basistype zijn opgelegd. De volgende declaratie is dus niet toegelaten:

```
suptype ILLEGALE_INDEX is NIET_NEGATIEF range -1 .. 1;
```

Het waardenbereik van het type NIET_NEGATIEF wordt hier immers overschreden en een foutmelding tijdens de verwerking zal hiervan het gevolg zijn. Ook subtypen zonder beperkingen zijn mogelijk. In dat geval is het subtype een synoniem voor het basistype:

```
subtype NON_NEGATIEF is NIET_NEGATIEF;
```

We kunnen ook subtypen van subtypen declareren (en voor afgeleide typen geldt dit ook). Voorbeeld:

```
subtype GROOT is NIET_NEGATIEF range 0 .. 1_000_000;
subtype KLEIN is GROOT          range 0 .. 10;
```

Tot de attributen van het basistype kan toegang worden verkregen met behulp van het attribuut BASE (dit kan op elk type worden toegepast):

```
INDEX'BASE'LAST
```

Bovenstaande grootheid heeft de waarde INTEGER'LAST. Het BASE attribuut moet altijd, zoals in bovenstaand voorbeeld, worden gebruikt tesamen met een ander attribuut.

Afgeleide typen. *Afgeleide typen* definiëren in tegenstelling tot subtypen wel afzonderlijke typen. Zij worden gebruikt als wij in onze probleemformulering verschillende objecten nodig hebben met dezelfde structuur. Wij kunnen bijvoorbeeld objecten nodig hebben van het type MASSA en van het type GEWICHT. Beide kunnen als onderliggende structuur het FLOAT type hebben, maar het kan nodig zijn ze afzonderlijk te behandelen. Beide typen kunnen in Ada als volgt worden gedeclareerd:

```
type MASSA    is new FLOAT;
type GEWICHT  is new FLOAT;
```

Zowel MASSA als GEWICHT zijn nu afgeleid van het type FLOAT, dat wel het oudertype wordt genoemd. Er kunnen nu objecten worden gedeclareerd van het type MASSA en van het type GEWICHT, maar zij kunnen niet zonder meer tesamen in expressies voorkomen. De waarde van een object van het type MASSA kan bijvoorbeeld niet zonder meer bij die van een object van het type GEWICHT worden opgeteld.

Een afgeleid type heeft alle eigenschappen van zijn ouder, zoals diens operaties, attributen en waardenverzameling. Een afgeleid type behoort dus tot dezelfde klasse als zijn ouder. Ook bij afgeleide typen is het weer mogelijk beperkingen toe te voegen, zoals bij subtypen:

```
type BUDGET is new FLOAT range 0.0 .. 12_000.0;
```

BUDGET heeft nu alle eigenschappen van zijn ouder FLOAT, met daaraan toegevoegd de beperking op het waardebereik. BUDGET objecten mogen niet zomaar met andere FLOAT objecten worden gemengd, maar er is wel een expliciete typeconversie mogelijk tussen een afgeleid type en zijn ouder. Bekijk eens de volgende declaraties:

```
SOFTWARE_BUDGET : BUDGET;
TOTAAL_BUDGET   : FLOAT;
```

Via de volgende instructie kan nu SOFTWARE_BUDGET naar het type van TOTAAL_BUDGET worden geconverteerd:

```
TOTAAL_BUDGET := FLOAT(SOFTWARE_BUDGET);
```

Er kan een afgeleid type van elk type worden gemaakt. Wel moet de definitie van de ouder volledig zijn uitgewerkt, voordat creatie van een afgeleid type toegelaten is. Afgeleide typen van onvolledige typedefinities en van private typen zijn dus niet toegelaten (tenzij in dit laatste geval ook het private deel van het pakket is beschreven).

Een afgeleid type krijgt alle operaties van zijn ouder mee. Dat betekent in het geval van een voorgedefinieerd type, alle voorgedefinieerde operaties. Voor gebruiker-gedefinieerde typen geldt dat een afgeleid type ook alle zichtbare subprogramma's meekrijgt, die parameters hebben of resultaten van het type van de ouder of van één van diens andere subtypen. Ook subprogramma's die alleen voor een bepaald afgeleid type gelden zijn mogelijk:

```
type DATUM is ...
function "+" (D_1 : in DATUM;
              D_2 : in DATUM) return DATUM is ...
procedure JULIAANS(D     : in  DATUM;
                   GETAL : out INTEGER) is ...
type MIJN_DATUM is new DATUM;
```

Op het moment van de declaratie van het afgeleide type MIJN_DATUM worden ook de subprogramma's "+" en JULIAANS overgenomen. Objecten van het type MIJN_DATUM kunnen gebruik maken van deze afgeleide operaties, maar als we na de declaratie van MIJN_DATUM opnieuw subprogramma's voor DATUM zouden formuleren, dan zouden die operaties niet als afgeleid voor MIJN_DATUM gelden.

8.3 Declaraties



We kunnen de abstracties uit ons probleemgebied in Ada dus met behulp van datatypen omschrijven, maar deze typen zijn slechts blauwdrukken. We moeten vervolgens objecten declareren van deze typen, die door onze programma's kunnen worden gemanipuleerd. We hebben deze objectdeclaraties al herhaalde malen gezien in het voorafgaande, maar er zijn nog enkele aspecten die nader moeten worden belicht.

Objecten worden ingevoerd in het declaratiegedeelte van het programma, of via de formele parameters van subprogramma's of generieke programma-eenheden. Steeds moeten we expliciet het type van een object aangeven als we een object declareren. Voorbeelden:

```
AFSTAND  : FLOAT;
ANTWOORD : CHARACTER;
GETAL    : INTEGER;
CIJFERS  : array (1 .. 100) of FLOAT;
```

Het type van CIJFERS heeft zelf geen naam: hier is sprake van een zogenaamd anoniem datatype. Een kenmerk van een goed programmeerstijl is dat elk type een naam heeft; in bepaalde gevallen is het invoeren van een anoniem type echter noodzakelijk. Tijdens de objectdeclaratie kan een beperking of randvoorwaarde worden toegevoegd. Dit is in de volgende voorbeelden het geval:

```
NAAM        : STRING(1 .. 40);
ONDERGRENS  : INTEGER range -10 .. -1;
```

Hier geldt als algemene stijlregel, dat het beter is in deze gevallen een subtype te creëren.

Ada is een taal met sterke typering. Dit betekent dat objecten van verschillende typen niet binnen één expressie willekeurig door elkaar kunnen worden gebruikt. Type equivalentie in Ada ontstaat alleen door expliciete benoeming; alleen als aan twee objecten hetzelfde type is verbonden, zijn zij ook inderdaad van hetzelfde type. Een voorbeeld:

```
type AFSTAND is digits 4;
type LENGTE  is digits 4;
BREEDTE : LENGTE;
MAAT    : AFSTAND;
```

MAAT en BREEDTE hebben niet hetzelfde type, hoewel de structuur op een lager niveau dezelfde is. Zo blijft het abstractieniveau gehandhaafd, ook als objecten qua structuur overeenkomstige typen bezitten.

Het is ook mogelijk aan objecten beginwaarden 'bij gebrek aan beter' (by default) toe te kennen:

```
MAAT : AFSTAND := 0.0;
```

Objecten hebben niet automatisch default waarden (de enige uitzondering zijn access objecten, met als beginwaarde **null**). De programmeur moet overigens alle objecten een beginwaarde toekennen (initialiseren). Als we een object proberen te gebruiken voordat dit een waarde kreeg, dan is het programma onjuist, maar het effect is implementatie-afhankelijk. De mogelijkheid is echter altijd aanwezig dat de uitzonderingstoestand PROGRAM_ERROR optreedt als we trachten een object te benaderen voordat dit geïnitialiseerd is.

Constante objecten worden gedeclareerd op dezelfde wijze als andere objecten, onder toevoeging van het gereserveerde woord **constant**. In het geval van numerieke constanten kan de typenaam worden weggelaten:

```
EERSTE_MAAND : constant MAAND_NAAM := JANUARI;
PI           : constant            := 3.141_592_65;
DOORSNEDE    : constant            := 4;
```

Getallen worden automatisch geconverteerd naar het bijbehorende door de gebruiker gedefinieerde type, afhankelijk van de context van het gebruik. Er kunnen ook meerdere objecten via één declaratie worden gedefinieerd:

```
I,J,K : INTEGER;
```

Het gebruik van deze mogelijkheid wordt in verband met de onderhoudbaarheid afgeraden. Overzichtelijker is:

```
I : INTEGER;
J : INTEGER;
K : INTEGER;
```

Oefeningen

*1. Op welke wijze maakt het typemechanisme in Ada, data-abstractie en beperkte toegankelijkheid van gegevens (information hiding) mogelijk?

2. Wat wordt door middel van een type aangegeven?

3. Formuleer de declaraties voor de volgende scalaire typen:
   (a) Een TELLER met alleen negatieve waarden.
   (b) Een COEFFICIENT met 12 significante cijfers.
   (c) Een METING met een grofheid van 0.1 centimeter.
   (d) Zie (c), maar nu met een bereik van 0.0 tot 10.0 meter.
   (e) De KLEUREN_VAN_DE_REGENBOOG.

4. Kan een typedefinitie worden gegeven voor een grootheid die alleen oneven gehele waarden mag aannemen? Licht uw antwoord toe.

5. Schrijf declaraties voor de volgende samengestelde typen:
   (a) Een array van 10 COEFFICIENTen.
   (b) Een niet begrensd array van METINGen, geïndexeerd met de KLEUREN_VAN_DE_REGENBOOG.
   (c) Zie (b), maar nu met een index beperkt tot de eerste drie kleuren.
   (d) Een PERSOONS_RECORD, waarin LEEFTIJD, GEWICHT, LENGTE en NAAM.
   (e) Voeg aan (d) een variant gedeelte toe. Voeg voor een discriminant KIND de waarden NAAM_MOEDER en NAAM_VADER toe, en voor een discriminant VOLWASSENE de waarde BEROEP.

6. Declareer een access type dat naar PERSOONS_RECORD kan verwijzen.

7. Verander in de declaratie van 5(e) het type van NAAM_MOEDER en van NAAM_VADER in het access type uit opgave 6.

8. Creëer drie PERSOONS_RECORDs en laat twee ervan als ouders naar de derde verwijzen, onder gebruikmaking van de declaratie uit opgave 7.

*9. Als we een pakket COMPLEX zouden declareren, met een private type GETAL, welke operaties zouden dan op objecten van dit type toegelaten zijn? Moet het type 'private' of 'limited private' zijn?

10. Formuleer de declaraties voor de volgende subtypen en afgeleide typen:
    (a) Een subtype van COEFFICIENTen met slechts 7 significante cijfers.
    (b) Een subtype van (a) met als bereik alleen positieve waarden.
    (c) Zie (a), maar nu via een afgeleid type.
    (d) Creëer uit 5(e) een subtype beperkt tot KIND.
    (e) Zie (d), maar gebruik nu een afgeleid type.

11. Declareer een object van elk type uit opgave 10 en ken elk object een beginwaarde toe.

9 HET TWEEDE PROBLEEM: DATABASE BENADERING

Het bouwen van een systeem voor het opvragen van gegevens uit een verzameling van onderling logisch samenhangende gegevensbestanden (een database) kan niet worden beschouwd als de ontwikkeling van een 'embedded', dat wil zeggen in een groter systeem ingebouwd, systeem. Tot nu toe werd een dergelijk database benaderingssysteem meestal in COBOL of een soortgelijke taal ontwikkeld, maar ook voor dit soort toepassingen is Ada geschikt. Trouwens, veel grote 'embedded systems' bevatten als deelsysteem ook programmatuur voor database benadering.

In dit hoofdstuk wordt de ontwikkeling van een database benaderingssysteem behandeld. We zullen de oplossing met behulp van onze object-gerichte ontwerpmethode uitwerken tot en met de definitie van de communicatiepatronen (interfaces) tussen de programma-eenheden. Om de oplossing volledig uit te werken zijn nog wat nieuwe gereedschappen nodig. Daartoe wordt in de hoofdstukken 10 en 11 nader ongegaan op subprogramma's en programma-instructies en vervolgens voltooien we de oplossing van dit tweede probleem in hoofdstuk 12.

9.1 Definieer Het Probleem



De STONEMAN specificaties vereisten de ontwikkeling van een Ada programmeeromgeving (APSE: Ada Programming Support Environment) ter ondersteuning van de totale levenscyclus van een software-project (zie hoofdstuk 3). APSE bepaalt in feite de wijze waarop de gebruiker tegen een computersysteem aankijkt. De programmeeromgeving vormt een overkoepeling van het besturingssysteem (operating system) en bevat het instrumentarium voor het ontwikkelen van systemen in Ada, zoals compilers (vertalers), editors (tekstverwerkers) en mogelijkheden voor configuratiebeheer. In tegenstelling tot een traditioneel operating system, maakt een programmeeromgeving als APSE het mogelijk het project te beheersen tijdens de probleemanalyse, in de ontwerpfase, tijdens de programmering, het testen, het in bedrijf zijn en bij onderhoud.

Hiertoe wordt gebruik gemaakt van de APSE database, waarin alle informatie over de ontwikkeling van een project wordt opgeslagen, zoals verschillende versies van programma-eenheden en documentatie daarvan. Op die manier kan een programmeur die een programma-eenheid creëert de overige bij het project betrokkenen daarvan op de hoogte stellen. Ook alle informatie over wijzigingen en testresultaten worden in de database opgeslagen ten behoeve van het configuratiebeheer (dat wil zeggen het bijhouden van versies van het systeem en het administreren van gegevens over wijzigingen).

Via de database kan de projectleider of de systeembeheerder gemakkelijker inzicht krijgen in de voortgang en heeft hij een betere greep op het project. De beheerder kan bijvoorbeeld gegevens verzamelen over het percentage gereedgekomen modules, of kan een 'baseline' willen vaststellen voor het testen van een versie van het systeem. Het werken met een *baseline* of operatiebasis bestaat uit het omschrijven van een bepaalde toestand tijdens de systeemontwikkeling (bijvoorbeeld het gereedkomen van het technisch ontwerp) en het bevriezen van de werkzaamheden, zodra die toestand bereikt wordt, zodat de voortgang en de kwaliteit van de ontwikkeling vanuit een stabiele situatie kan worden beoordeeld. Met behulp van APSE kunnen de daartoe benodigde acties deels automatisch worden uitgevoerd.

We zullen in dit stadium ons ontwerp van een APSE database niet geheel kunnen voltooien, omdat we een aantal bijzonderheden van Ada's subprogramma's en programma-instructies nog niet hebben behandeld. Allereerst zullen we nu een systeem ontwerpen voor het opvragen van gegevens uit een database.

9.2 Ontwikkel Een Informele Oplossingsstrategie



Om te beginnen kijken we alleen vanuit het gezichtspunt van de gebruiker naar de database: hoe wil hij of zij deze kunnen benaderen? We streven daarbij naar logische gegevensonafhankelijkheid, dat wil zeggen, we ontwerpen onafhankelijk van mogelijke fysieke representaties. De bedoeling daarvan is dat, mocht het nodig zijn de structuur van de database op het implementatieniveau te wijzigen (bijvoorbeeld ter verbetering van de efficiëntie), op logisch niveau geen wijzigingen behoeven te worden aangebracht. Dit bevordert ook het lokaal houden van effecten van wijzigingen. Als nieuwe mogelijkheden later moeten worden toegevoegd betekent dit geen aantasting van de logische structuur.

Een database opvraagsysteem is passief, dat wil zeggen, het is niet mogelijk gegevens toe te voegen, te wijzigen, of te verwijderen. Het ontwerp van voorstellingswijzen voor de database elementen en van de functies voor het opvragen van gegevens is van later zorg; we zullen om te beginnen het principe van beperkt toegankelijk

maken van informatie ('information hiding') toepassen de de top-down ontwerpmethode gebruiken.

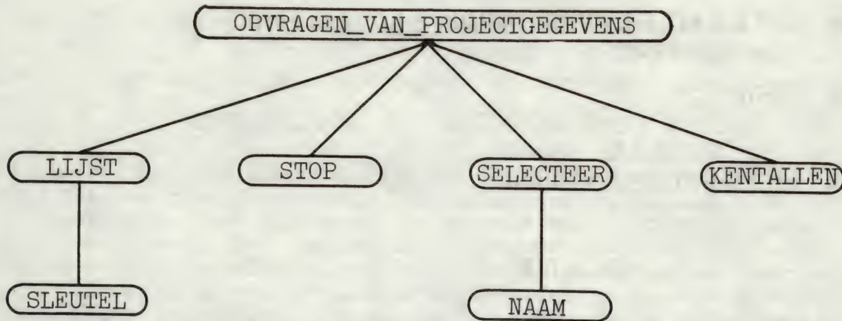
We formuleren voor onze objectgerichte ontwerpmethode nu eerst informeel de oplossingsstrategie:

> **Een database in APSE wordt gebruikt om de informatie op te slaan over elke programma-eenheid van een bepaald object. De over elke eenheid opgeslagen informatie omvat de naam van de eenheid, de programmeur, de voortgangsstatus, identificatie van de huidige versie en een verwijzing naar het bijbehorend specificatiedocument. De interactieve bewerkingen, waartoe een gebruiker opdracht kan geven zijn onder meer: het produceren van een lijst van projectgegevens (gesorteerd op programmeursnaam of op grond van basisspecificaties), het weergeven van alle gegevens over een gekozen programma-eenheid en het verzamelen van kentallen over de ontwikkelingsfase van alle programma-eenheden. Een opvraagsessie duurt totdat de gebruiker een verzoek om de sessie te beëindigen invoert.**

Het is duidelijk dat nog niet alle bijzonderheden van het probleem uitputtend zijn beschreven, maar toch is bovenstaande informele beschrijving acceptabel, omdat die bijzonderheden in dit stadium nog niet van belang zijn. We zullen verder het probleem tot een aanvaardbare omvang beperken door de volgende randvoorwaarden toe te voegen:

- Om een lijst te produceren voert de gebruiker het sorteercriterium in (de programmanaam of de specificatie-identificatie). Vervolgens wordt een op grond van dit criterium gesorteerde lijst van programma-eenheden gemaakt. Als de gebruiker bijvoorbeeld *naam* als sorteersleutel gebruikt, dient alle informatie alfabetisch te worden gesorteerd op programmanaam.
- Een bepaalde programma-eenheid wordt door de gebruiker geselecteerd door de naam van de eenheid op te geven. Vervolgens wordt een lijst met alle gegevens van die eenheid geproduceerd.
- Voor het produceren van kentallen dient de absolute en relatieve frequentie van de staten van ontwikkeling van alle eenheden te worden bepaald (de *status* kan zijn: ontwerpfase, coderingsfase, testfase en operationeel). Er moet dus een lijst, gesorteerd naar status worden geproduceerd, met bij elke status het aantal programma-eenheden en het aantal als percentage van het totaal.

Op grond van deze informele specificaties kan het systeem vanuit gebruikersstandpunt schematisch zoals in figuur 9-1 worden samengevat als een hiërarchisch opgebouwde verzameling commando's. Bij de verdere ontwikkeling van dit systeem zullen we nadere ontwerpbeslissingen nemen in verband met nu nog niet volledig bekende probleemaspecten.



Figuur 9-1 Hiërarchie van commando's voor OPVRAGEN_VAN_PROJECTGEGEVENS

9.3 Formaliseer De Strategie



Nu dient de informele strategie formeel in Ada te worden beschreven.

Identificeer de objecten en hun kenmerken

We schrijven de informele strategie nu nog eens op en onderstrepen de objecten die voor ons systeem van belang zijn:

> Een database in APSE wordt gebruikt om de informatie op te slaan over elke programma-eenheid van een bepaald project. De over elke eenheid opgeslagen informatie omvat de naam van de eenheid, de programmeur, de voortgangsstatus, identificatie van de huidige versie en een verwijzing naar het bijbehorend specificatiedocument. De interactieve bewerkingen, waartoe de gebruiker opdracht kan geven zijn onder meer: het produceren van een lijst van projectgegevens (gesorteerd op programmeursnaam of op grond van basisspecificaties), het weergeven van alle gegevens over een gekozen programma-eenheid en het verzamelen van kentallen over de ontwikkelingsfase van alle programma-eenheden. Een opvraagsessie duurt totdat de gebruiker een verzoek om de sessie te beëindigen invoert.

Bij het bestuderen van de onderstreepte objecten moeten we gebruik maken van onze kennis over het probleem. Er zijn bijvoorbeeld objecten onderstreept, zoals APSE, opvraagsessie en gebruiker, die wel degelijk tot het probleem behoren, maar die niet via een abstractie naar de oplossingsruimte hoeven te worden afgebeeld. Ook moeten we benamingen identificeren, die naar hetzelfde object verwijzen, zoals eenheid, informatie en database. Dit leidt tot de volgende lijst van relevante objecten:

- OPVRAAG_BEWERKINGEN
  - OPDRACHT

- PROJECT
  - EENHEID_INFORMATIE DATA_BASE
    - EENHEID_NAAM
    - PROGRAMMEUR
    - STATUS
    - VERSIE
    - SPECIFICATIE

Het gaat dus in ons systeem maar om twee primaire objecten: OPVRAAG_BEWERKINGEN en PROJECT. Beide hebben één of meer subobjecten. Het object PROJECT bestaat uit EENHEID_INFORMATIE (datgene waarvoor wij de database gebruiken) en de DATA_BASE zelf. Te zijner tijd (nu is het daarvoor nog te vroeg) zullen we een voorstellingswijze voor alle objecten ontwikkelen.

Identificeer de bewerkingen op de objecten

Nu moeten we de bewerkingen vaststellen die op onze objecten mogelijk zijn. We onderstrepen ze in onze informele strategie:

> **Een database in APSE wordt gebruikt om de informatie op te slaan over elke programma-eenheid van een bepaald project. De over elke eenheid opgeslagen informatie omvat de naam van de eenheid, de programmeur, de voortgangsstatus, identificatie van de huidige versie en een verwijzing naar het bijbehorend specificatiedocument. De interactieve bewerkingen, waartoe de gebruiker opdracht kan geven zijn onder meer: het produceren van een lijst van projectgegevens (gesorteerd op programmeursnaam of op grond van basisspecificaties), het weergeven van alle gegevens over een gekozen programma-eenheid en het verzamelen van kentallen over de ontwikkelingsfase van alle programma-eenheden. Een opvraagsessie duurt totdat de gebruiker een verzoek om de sessie te beëindigen invoert.**

De mogelijke bewerkingen zijn dus:

```
OPVRAAG_BEWERKINGEN
    OPDRACHT
        VERZOEK

PROJECT
    EENHEID_INFORMATIE          DATA_BASE
        VERZAMEL_KENTALLEN
        PRODUCEER_LIJST
        STOP
        SELECTEER_EENHEID
```

We gaven de objecten en operaties weer zoveel mogelijk namen, overeenkomend met de beschrijving in de informele strategie.

Zelfstandige naamwoorden werden objecten en werkwoorden werden operaties. De operatie VERZOEK is enigszins kunstmatig ingevoerd, omdat de keuze daarvan de leesbaarheid van de uiteindelijke oplossing blijkt te bevorderen. Hierboven hebben we de operaties gerangschikt bij de objecten, waarop zij van toepassing zijn, maar in het vervolg van de oplossing kan blijken dat een hergroepering nodig is om de begrijpelijkheid en onderhoudbaarheid te vergroten.

Tot nu toe bevinden we ons nog steeds op het hoogste abstractieniveau; in het vervolg zullen we lokale objecten en bijbehorende operaties toevoegen.

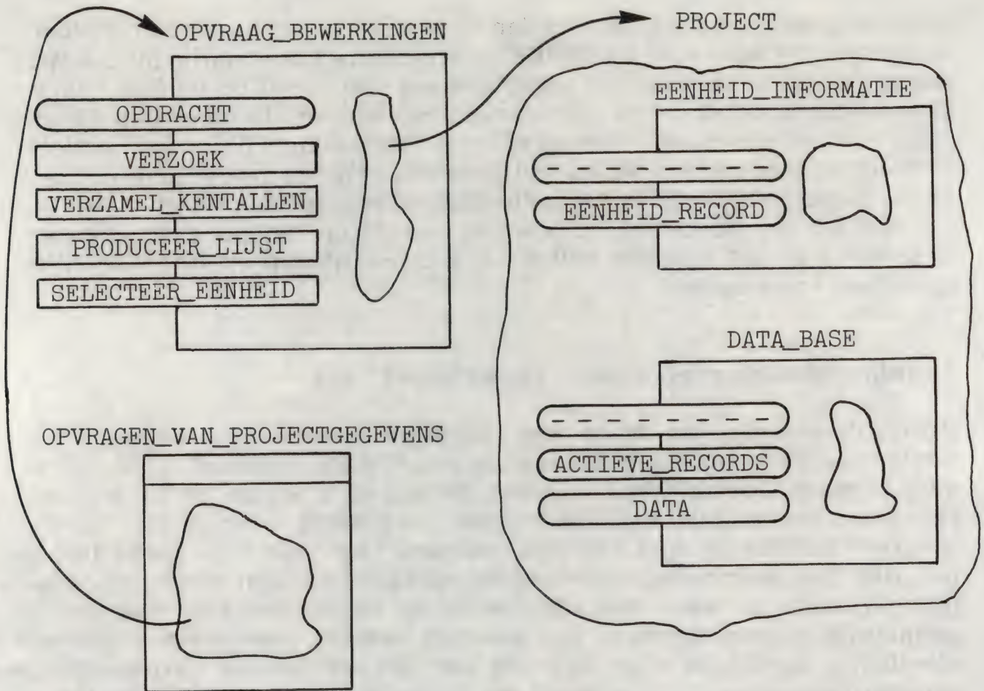
Stel de communicatiepatronen (interfaces) vast

Nu we de objecten en de daarop mogelijke bewerkingen kennen, moeten we de samenhang tussen de grootheden, waaruit onze oplossing bestaat, formuleren. Voordat dit mogelijk is moeten we echter eerst nog wat nadere bijzonderheden over APSE vermelden.

De STONEMAN specificaties voorzien nog in andere gereedschappen dan een eenvoudige opvraagmogelijkheid uit een database, namelijk gereedschap voor het administreren en controleren van verschillende systeemversies. Het gebruik van dit gereedschap gebeurt steeds via de database en daaraan kunnen ook nieuwe gereedschappen worden toegevoegd. De gebruikers hebben gewoonlijk niet direct toegang tot de database; alle operaties worden geregeld via de programmeeromgeving. De gereedschappen binnen de programmeeromgeving moeten dus zijn afgestemd op de databasestructuur, maar de gebruiker hoeft hierin geen inzicht te hebben.

Ter verduidelijking het volgende voorbeeld. Stelt u zich voor dat u naar een grote bibliotheek gaat voor literatuuronderzoek. Het zoeken naar gegevens over een bepaald onderwerp kan zowel met de hand als met behulp van een geautomatiseerd systeem gebeuren. Veronderstel dat het niet mogelijk is de boeken zelf te halen, maar dat u de titels aan een bibliothecaris moet opgeven, die ze vervolgens voor u opzoekt. Als nu een nieuwe mogelijkheid ter beschikking kwam voor het opzoeken van gegevens, zoals bijvoorbeeld microfiches, dan zou dit niet betekenen dat de organisatie van de bibliotheek veranderde, maar alleen ons interface (wijze van communicatie) met de bibliotheekinhoud verandert. En ook als de bibliothecaris alle boeken op een andere manier zou rangschikken, dan zou ons dat niet interesseren (tenzij we zelf de bibliothecaris waren natuurlijk), want vanuit logisch standpunt blijft de bibliotheek onveranderd.

Terug naar ons probleem: in figuur 9-2 is weergegeven hoe elk project kan worden gezien als een eenheid uit de APSE bibliotheek. Elk project is verbonden met de DATA_BASE door een datawoordenboek (data dictionary of *schema*), dat de structuur van de EENHEID_INFORMATIE aangeeft. Merk op dat de gegevens zelf gescheiden worden behandeld van de beschrijving van hun structuur. Zo heeft de gebruiker wel toegang tot de beschrijvingen, maar blijft de informatie in de DATA_BASE beschermd. Ook is het mogelijk dat er nog



Figuur 9-2 Ontwerp voor OPVRAGEN_VAN_PROJECTGEGEVENS

meer pakketten zijn, zoals EENHEID_INFORMATIE, die weer een ander aspect van de DATA_BASE beschrijven. Daar die echter buiten de huidige toepassing vallen, behoeven we ons daar niet mee bezig te houden. In figuur 9-2 hebben we verder niet alle eenheden aangegeven die door de pakketten EENHEID_INFORMATIE en DATA_BASE naar buiten worden gebracht (zie de met een stippellijn gevulde rechthoekjes), maar alleen de belangrijkste. Geen van de eenheden maakt operaties van buiten toegankelijk, alleen data worden geëxporteerd (dit wordt aangegeven door de afgeronde hoeken van de rechthoekjes).

Voor onze toepassing gaat het om het OPVRAAG_BEWERKINGEN pakket. Dit verschaft het instrumentarium voor de oplossing van ons probleem. Ingekapseld in deze eenheid zitten de operaties die op de projectgegevens kunnen worden toegepast. Omdat OPVRAAG_BEWERKINGEN ook bij andere PROJECTen van pas kan komen, zou het pakket als generiek kunnen worden gedefinieerd (zie hoofdstuk 14). Om de oplossing wat eenvoudiger te houden zullen we dat niet doen, maar veronderstellen dat OPVRAAG_BEWERKINGEN alleen voor ons eigen PROJECT zal worden gebruikt. In figuur 9-2 is een pijl getekend van OPVRAAG_BEWERKINGEN naar PROJECT, om de onderlinge afhankelijkheid van deze twee grootheden aan te geven. Net

zoals bij ons eerste probleem (hoofdstuk 7) zullen we een hoofdprogramma kiezen voor onze oplossingsmethode, dat we hier OPVRAGEN-_VAN_PROJECT_GEGEVENS noemen. Deze programma-eenheid zal gebruik maken van het pakket OPVRAAG_BEWERKINGEN, zoals in figuur 9-2 is aangegeven. Andere mogelijke pakketten, zoals bijvoorbeeld een pakket voor dynamische bewerkingen (toevoegen, wijzigen of verwijderen van gegevens) betrekken wij niet bij ons huidige probleem.

In Ada is het mogelijk de pakketten DATA_BASE en EENHEID-_INFORMATIE in te kapselen binnen een groter pakket, PROJECT genaamd. Het hoofdprogramma moet wel toegang hebben tot het pakket OPVRAAG_BEWERKINGEN, maar de DATA_BASE hoeft voor het hoofdprogramma niet direct toegankelijk te zijn.

Op het huidige abstractieniveau, als ontwikkelaars van de oplossingsmethode, hoeven we alleen maar te weten hoe de gereedschappen van OPVRAAG_BEWERKINGEN en hoe een EENHEID_INFORMATIE gebruikt moeten worden. Een Ada-pakket bestaat uit twee gedeelten: de specificatie en de romp of body (zie hoofdstuk 6). De specificatie geeft aan welke de van buitenaf zichtbare grootheden uit het pakket zijn (de geëxporteerde grootheden). De body vormt de uitwerking van de mechanismen van de operaties die in de specificatie worden omschreven. Omdat de specificatie een overeenkomst op logisch niveau met de gebruiker vormt, dienen we nu eerst dit communicatiepatroon te definiëren en zullen we de bodies later uitwerken. Het schrijven van Ada programmagedeelten, die de van buitenaf zichtbare delen van OPVRAAG_BEWERKINGEN, EENHEID_INFORMATIE en DATA_BASE definiëren, is nu dus onze eerstvolgende taak.

EENHEID_INFORMATIE wordt een pakket binnen een groter pakket, PROJECT genaamd. Het eerste stukje Ada code ziet er dan als volgt uit:

```
package PROJECT is
   package EENHEID_INFORMATIE is
      . . .
   end EENHEID_INFORMATIE;
   package DATA_BASE is
      . . .
   end DATA_BASE;
end PROJECT;
```

Het PROJECT pakket bestaat dus uit een collectie andere pakketten. De bodies van het pakket EENHEID_INFORMATIE en het pakket DATA_BASE zijn vooralsnog leeg. Bedenk verder dat EENHEID-_INFORMATIE eerst moet worden gedefinieerd, omdat DATA_BASE van EENHEID_INFORMATIE gebruik maakt.

Bij het verder uitwerken van het EENHEID_INFORMATIE pakket maken we gebruik van de mogelijkheden tot het creëren van elementaire datatypen, die we in hoofdstuk 8 besproken hebben. Opnieuw gebruiken we de op de waargenomen werkelijkheid gebaseerde informele strategie als model voor onze oplossing:

```
package EENHEID_INFORMATIE is
  type NAAM_TYPE              is new STRING(1..20);
  type PROGRAMMEUR_TYPE       is new STRING(1..20);
  type REFERENTIE;
  type REFERENTIE_TOEGANG is access REFERENTIE;
  type REFERENTIE             is record
                                 DOCUMENT : STRING(1..80);
                                 PAGINA   : POSITIVE;
                                 VOLGENDE : REFERENTIE_TOEGANG;
                              end record;
  type STATUS_TYPE       is (ONTWERP,CODE,TEST,OPERATIONEEL);
  type VERSIE_TYPE       is range 0 .. 99;
  type EENHEID_RECORD is record
                            EENHEID_NAAM : NAAM_TYPE;
                            PROGRAMMEUR  : PROGRAMMEUR_TYPE;
                            SPECIFICATIE : REFERENTIE_TOEGANG;
                            STATUS       : STATUS_TYPE;
                            VERSIE       : VERSIE_TYPE;
                         end record;
end EENHEID_INFORMATIE;
```

Voor de uitwerking van dit pakket hadden we nogal wat verschillende typen nodig. Om de leesbaarheid te bevorderen hebben we de componenten in alfabetische volgorde geplaatst. Om te beginnen definieerden we enkele van het type STRING afgeleide typen; immers grootheden als EENHEID_NAAM en PROGRAMMEUR mogen niet onderling verward worden. Voor de SPECIFICATIE informatie gebruikten we eerst een onvolledig datatype, en vervolgens een recorddeclaratie. We kozen voor de structuur van de verbonden lijst ('linked list'), omdat op dit moment niet duidelijk is hoveel SPECIFICATIEs er bij een bepaalde eenheid kunnen horen. Dan volgen twee gebruikergedefinieerde typen en wel een enumeratietype en een waardenbereik voor geheeltallige grootheden. Tenslotte bevat EENHEID_RECORD de gegevens, zoals we die in de informele strategie omschreven.

Het pakket bestaat in feite uit een collectie declaraties en het exporteert een aantal typen. Specifieke elementen uit het pakket kunnen worden benaderd via de puntnotatie:

```
PROJECT.EENHEID_INFORMATIE.STATUS_TYPE
PROJECT.EENHEID_INFORMATIE.EENHEID_RECORD
```

De gebruiker van OPVRAGEN_VAN_PROJECTGEGEVENS behoeft geen toegang te hebben tot de DATA_BASE zelf, maar wij als implementators van de OPVRAAG_BEWERKINGEN hebben de DATA_BASE wel nodig. De uitwerking van de DATA_BASE luidt:

```
package DATA_BASE is
   use EENHEID_INFORMATIE
   MAXIMUM_RECORDS        : constant := 100;
   type RECORD_INDEX      is range 0 .. MAXIMUM_RECORDS;
   type PROJECT_RECORDS is array (RECORD_INDEX)
                                   of EENHEID_RECORD;
   ACTIEVE_RECORDS        : RECORD_INDEX;
   DATA                   : PROJECT_RECORDS;
end DATA_BASE;
```

De use declaratie werd hier gebruikt om de leesbaarheid te bevorderen. Elementen uit EENHEID_INFORMATIE kunnen nu direct worden benaderd en daarom kan bijvoorbeeld EENHEID_RECORD worden gebruikt. Als de use declaratie niet was gebruikt, dan hadden we moeten schrijven:

```
EENHEID_INFORMATIE.EENHEID_RECORD
```

Om de oplossing eenvoudig te houden zijn we uitgegaan van een statisch aantal DATA elementen. Bij een praktische toepassing zou een dynamisch bereik hier geschikter zijn. We veronderstellen verder dat de DATA objecten beginwaarden hebben gekregen (bijvoorbeeld in de nadere uitwerking van de PROJECT pakket body).

We beschrijven nu het toegankelijke gedeelte van het pakket OPVRAAG_BEWERKINGEN. Deze grootheid bestaat uit een verzameling logisch samenhangende subprogramma's en levert de bewerkingen die de gebruiker op de database kan toepassen. Op grond van de bewerkingen die we bij de uitwerking van onze oplossing afleidden, komen we tot het volgende pakket:

```
with PROJECT;
use PROJECT;
package OPVRAAG_BEWERKINGEN is
  type OPDRACHT is (VERZAMEL_KENTALLEN,PRODUCEER_LIJST,
                    STOP,SELECTEER_EENHEID);
  function VERZOEK return OPDRACHT;
  procedure VERZAMEL_KENTALLEN;
  procedure PRODUCEER_LIJST;
  procedure SELECTEER_EENHEID;
end OPVRAAG_BEWERKINGEN;
```

De volgende stap zou het uitwerken van de procedures zijn, die in de pakketspecificatie worden vermeld. Maar evenmin als in hoofdstuk 7 hebben we het benodigde instrumentarium tot onze beschikking om de algoritmen volledig te formuleren. We stellen de verdere uitwerking daarom uit tot hoofdstuk 12. In het volgende hoofdstuk zal het onderwerp subprogramma's verder worden uitgediept. De daarop volgende hoofdstukken gaan in op het gebruik van programma-instructies ter bestudering van de uitvoering van algoritmen. Gewapend met deze instrumenten zullen we ons database benaderingsprobleem opnieuw te lijf gaan.

Oefeningen

1. Schrijf de specificatie voor een databasegereedschap, SORTEER_BEWERKINGEN genaamd. Ga er vanuit dat dit pakket de bewerkingen SORTEER_OPLOPEND en SORTEER_AFLOPEND levert, waarbij de waarden van PROGRAMMEUR als sleutel worden gebruikt. Voeg ook een geschikt OPDRACHT type toe.

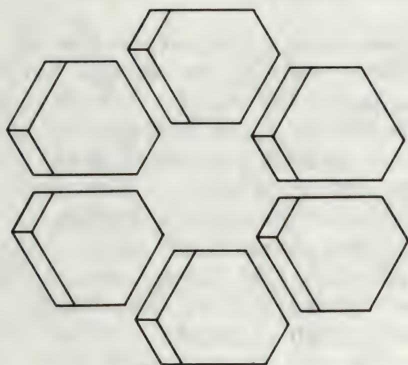
2. Schrijf de specificatie voor een databasegereedschap RAPPORT_GENERATOR. Bewerkingen zijn: RAPPORTAGE_OVER_LANGZAME_PROGRAMMEURS en RAPPORTAGE_OVER_UITZONDERLIJKE_PROGRAMMEURS. Voeg ook hier een geschikt OPDRACHT type toe.

3. Waarom zou het gevaarlijk zijn de gebruiker direct tot de DATA_BASE toe te laten in plaats van via bepaald gereedschap?

*4. Waarom denkt u dat wij ervoor kozen de bewerkingen in OPVRAAG_BEWERKINGEN zonder parameters te formuleren?

Pakket 4

ALGORITMEN EN BESTURING



Voeg de daad bij het woord,
het woord bij de daad ...

Shakespeare
Hamlet [1]

10 SUBPROGRAMMA'S

Het doel van ieder computerprogramma is om een bepaalde hoeveelheid werk te verrichten, hoe ook gemeten. We zagen al eerder dat programma's zowel beschrijvend als gebiedend zijn; de gebruiker ziet een programma als een verzameling gegevensobjecten die elkaar beïnvloeden via een rij sequentiële of parallelle acties. Idealiter vormen deze acties een directe afbeelding van het proces uit de werkelijkheid; een programma voor de besturing van een elektronische oven dient bijvoorbeeld operaties te kennen als: ZET_VERHITTINGSELEMENT_AAN, LEES_TEMPERATUURVOELER en LEES_TEMPERATUURINSTELLING. Als programma's op dit abstractieniveau kunnen worden geformuleerd wordt de begrijpelijkheid en onderhoudbaarheid van het systeem vergroot.

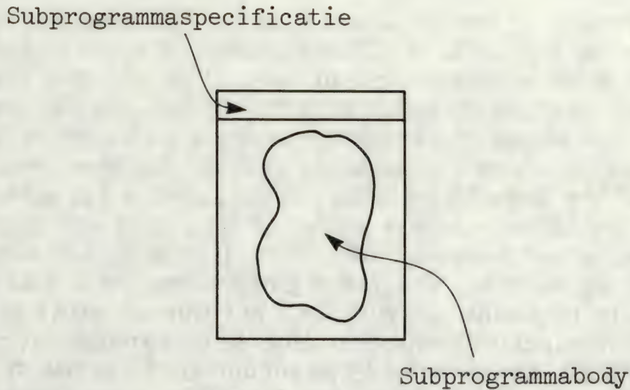
Nu kan geen enkele hogere programmeertaal alle operaties bevatten die maar mogelijk zijn. Talen beschikken daartoe gewoonlijk over een mechanisme om met behulp van elementaire instructies complexere te formuleren en aldus algoritmen te ontwikkelen. Dit mechanisme is het *subprogramma*. Op dezelfde manier als waarop abstracte datatypen kunnen worden opgebouwd uit elementairder typen, kunnen acties op een hoger abstractieniveau worden opgebouwd uit elementairder acties, gespecificeerd met behulp van elementaire instructies. De uitwerking van de specificaties tot acties op het hogere abstractieniveau naar een aantal elementairder instructies kan tot een latere fase worden uitgesteld. Bij eerdere voorbeelden gebruikten we al een aantal malen subprogramma's, maar nu zullen we hun structuur en toepassingsmogelijkheden in detail gaan bekijken.

10.1 Subprogramma's In Ada



Subprogramma's vormen de elementaire grootheden in Ada, die door de computer kunnen worden uitgevoerd. Zij komen voor in twee verschijningsvormen:

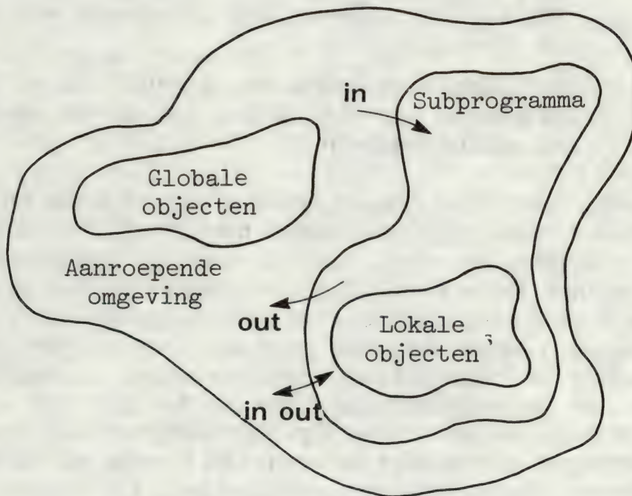
- procedures
- functies



Figuur 10-1 Symbool voor een Ada subprogramma

Een *procedure* geeft een naam aan een verzameling instructies, die tesamen een actie op een hoger abstractieniveau bewerkstelligen. *Functies* doen hetzelfde, maar leveren daarbij altijd een berekende waarde als resultaat. Subprogramma's worden *gedefinieerd* via de declaratie en *in werking gesteld* via een subprogramma-aanroep. Subprogramma's kunnen overal gedeclareerd worden, waar een objectdeclaratie toegelaten is, dus in een pakketspecificatie en in het declaratiegedeelte van een blok, een subprogramma body, een taak body of een pakket body.

In hoofdstuk 6 tekenden we het symbool voor een Ada subprogramma; in figuur 10-1 herhalen we dat nog eens. Subprogramma's



Figuur 10-2 Model voor een Ada subprogramma

kunnen echter ook nog op een andere manier bekeken worden en dat is geillustreerd in figuur 10-2. Uit dit plaatje is op te maken dat een subprogramma door de omgeving van waaruit de aanroep plaats vindt als een logische eenheid beschouwd kan worden. Het subprogramma kan daarbij lokale objecten bevatten, die voor de buitenwereld verborgen zijn. Anderzijds kunnen bepaalde globale objecten voor het subprogramma zelf wel bereikbaar zijn. Communicatie via globale grootheden buiten het subprogramma verhoogt de graad van koppeling tussen programma-eenheden en dit wordt daarom in het algemeen niet gezien als passend bij een goede programmeerstijl (zie nogmaals hoofdstuk 4). In bepaalde gevallen kan het binnen grote real-time systemen toch noodzakelijk zijn van globale grootheden gebruik te maken om efficiënte verwerking te bevorderen. Maar ook in die gevallen is het mogelijk de effecten van dergelijke verwijzingen naar globale grootheden via het Ada pakketmechanisme tot duidelijk aanwijsbare gedeelten van het systeem te beperken.

Beter is het via parameters met subprogramma's te communiceren en objecten van het gewenste type door te geven of te ontvangen. De objecten die bij een subprogramma-aanroep worden doorgegeven heten de *actuele* parameters; de namen die in de parameterlijst worden gebruikt bij de specificatie van het subprogramma, heten de *formele* parameters. Parameters kunnen op verschillende manieren worden doorgegeven:

- **in** — De actuele parameterwaarde wordt door het subprogramma gebruikt, maar kan er niet door gewijzigd worden.
- **out** — Het subprogramma genereert een waarde en kent deze toe aan een uitvoerparameter, maar maakt daarbij geen gebruik van de actuele waarde van die parameter.
- **in out** — Het subprogramma maakt gebruik van de actuele waarde en kan een nieuwe waarde aan de actuele parameter toekennen.

Modulariteit, abstractie, lokaal houden van effecten en beperkt toegankelijk maken van gegevens, de in hoofdstuk 4 besproken grondslagen voor goed ontwerp, kunnen door het gebruik van subprogramma's worden bevorderd. Oorspronkelijk, bij het gebruik van eerste generatie programmeertalen, werden subprogramma's voornamelijk gepropageerd omdat daarmee geheugen kon worden bespaard, doordat herhaling van identieke series instructies kon worden voorkomen. Dit is niet de gedachtengang die aan het gebruik van subprogramma's in Ada ten grondslag ligt: subprogramma's dienen directe afbeeldingen te zijn van de operaties, zoals we die binnen onze abstracte probleemoplossing formuleerden. De ontwerper moet niet uitgaan van de beperkingen, die hem door de binnen een programmeertaal beschikbare primitieve bouwstenen worden opgelegd, maar moet beginnen de bewerkingen die hij nodig heeft voor de

oplossing van zijn probleem te formuleren. Vervolgens kan hij subprogramma's gebruiken om de voor die bewerkingen benodigde mechanismen te implementeren. Als op deze manier van subprogramma's gebruik wordt gemaakt, dan wordt de programmeertaal uitbreidbaar; dat wil zeggen: aan de taal worden die uitdrukkingsmogelijkheden toegevoegd, die het de programmeur mogelijk maken zijn oplossingsmethode in termen van het probleemgebied te formuleren.

Subprogrammaspecificaties

Subprogramma's bestaan net als pakketten en taken uit een specificatiegedeelte en een body. Het specificatiegedeelte omschrijft de interface met de buitenwereld, dat wil zeggen de wijze waarop het subprogramma moet worden geactiveerd. Deze specificatie bestaat uit de naam van het subprogramma, gevolgd door nul of meer parameters. Omdat Ada een taal is, waarin alle typen expliciet moeten worden aangegeven, bevat de subprogrammaspecificatie ook een definitie van de typen van de gebruikte parameters. Als het subprogramma een functie is dan wordt ook het type van de door het subprogramma geretourneerde waarde aangegeven. Hier volgen een aantal voorbeelden van subprogrammaspecificaties:

```
procedure TEL_BLAADJES_IN_BINAIRE_BOOM;
procedure PUSH    (ELEMENT: in INTEGER; OP: in out BUFFER);
procedure ROTEER (PUNT: in out COORDINAAT; HOEK: in RADIALEN);
function  COS     (HOEK: RADIALEN) return FLOAT;
function  RANDOM                   return FLOAT;
function  "*"     (X,Y  : in MATRIX) return MATRIX;
```

De **in** modus voor parameters wordt bij 'default', dat wil zeggen bij 'verstek', dus als niets wordt gespecificeerd, aangenomen. HOEK in de functie COS is dus een invoerparameter. Terwille van de leesbaarheid is het aan te raden de modus wel altijd expliciet te vermelden. Bij procedures is de gemengde modus **in out** toegelaten; bij functies is alleen de **in** modus toegelaten; op die manier worden ongewenste bij-effecten ('side effects') voorkomen. Om een voorbeeld te geven: het zou een ongewenst bij-effect zijn als de functie COS ook de waarde van HOEK wijzigde, maar wegens de **in** modus kan dat dan ook niet. Bij-effecten (vaak veroorzaakt door benadering van een globale grootheid) zijn in de regel niet gewenst, maar er zijn gevallen waarin zij nodig zijn. Een RANDOM functie bijvoorbeeld, moet ook de globale startwaarde wijzigen, om telkens op grond van een vorige waarde een volgende pseudo random waarde te bepalen; deze doorgeefwaarde kan dus niet als **in** parameter worden gespecificeerd. Een tweede voorbeeld is het ophogen van een globale teller, telkens als een bepaalde programma-eenheid wordt benaderd; een gebruikelijke methode tijdens de testfase. Ook hier moet een globaal object worden gebruikt, maar steeds moeten we er in dergelijke

gevallen voor zorgen dat het bereik (de 'scope') van zo'n object zo beperkt mogelijk wordt gehouden en duidelijke documentatie is in deze gevallen van groot belang.

Het getuigt ook van een goede programmeerstijl om het aantal parameters per subprogramma klein te houden; alleen de gegevens, nodig en voldoende voor de correcte werking van het subprogramma moeten als parameters worden doorgegeven. Door het aantal parameters in de buurt van de in hoofdstuk 4 besproken Hrair-limiet te houden, wordt de leesbaarheid en begrijpelijkheid van de subprogrammadeclaratie vergroot. Door het gebruik van betekenisvolle en daardoor soms lange namen kunnen subprogrammadeclaraties meer dan één regel in beslag nemen. Door het gebruik van een goede layout kan de leesbaarheid in die gevallen toch redelijk blijven:

```
procedure ACTIVEER(PROCES      : in out PROCES_NAAM;
                   PRIORITEIT  : in     NATURAL;
                   WACHT       : in     TIJDSDUUR);
procedure PRINT   (KOPTEKST    : in     STRING;
                   CENTREREN   : in     BOOLEAN := TRUE;
                   PAGINEREN   : in     BOOLEAN := TRUE);
```

In het tweede voorbeeld kregen CENTREREN en PAGINEREN verstekwaarden. In het vervolg zullen we nog zien hoe het toekennen van verstekwaarden aan parameters de subprogramma-aanroep kunnen beïnvloeden.

Subprogramma's kunnen ook een naam krijgen als '*', dat wil zeggen een bepaald operatorsymbool. Dit operatorsymbool krijgt daardoor meer dan één betekenis; het wordt *overladen*. Alleen de operatoren, zoals in hoofdstuk 6 onder expressies besproken, kunnen op deze wijze gebruikt worden. (Een uitzondering vormt het samengestelde symbool '/=' (niet gelijk aan), dat altijd impliciet wordt gedeclareerd, zodra het symbool '=' (gelijk aan) wordt overladen.) Ada kent zelf alleen de wiskundige operaties op elementaire typen, zoals INTEGER en FLOAT (en de daarvan afgeleide numerieke typen), maar de programmeur kan de operatorsymbolen als functienamen gebruiken om operaties op eigen typen, zoals matrixvermenigvuldiging of rekenen met complexe getallen mogelijk te maken. In die gevallen kunnen de wiskundige eigenschappen van de aldus door de gebruiker gedefinieerde operaties niet meer door Ada worden gegarandeerd; het is dus van belang dat de gebruiker er zelf voor zorgt niet van de rekenkundige conventies af te wijken.

Als we het teken '*' herdefiniëren voor matrixvermenigvuldiging, dan is de vermenigvuldigingsoperator overladen: hetzelfde teken wordt voor verschillende doeleinden gebruikt. Zo heel bijzonder is dit niet, want in veel talen gebeurt dat impliciet en worden rekenkundige operatoren en bijvoorbeeld input/output procedures als READ en WRITE overladen voor gebruik met meer dan één type. In Pascal bijvoorbeeld zijn de volgende subprogramma-aanroepen toegelaten:

```
WRITELN(INDEX);
WRITELN("Tik uw naam in");
```

Dezelfde subprogrammanaam wordt hier voor twee verschillende doeleinden gebruikt: het uitvoeren van de waarde van een object en het uitvoeren van een tekst. Ook andere namen dan de operatortekens kunnen in Ada op deze manier overladen worden:

```
procedure ZET(LISTING     : in BOOLEAN);
procedure ZET(PIXEL       : in KLEUR; RAND : in out BUFFER);
procedure ZET(PRIORITEIT: in NATURAL);
procedure ZET(ADRES       : in NATURAL);
```

Een aanroep van een dergelijk overladen subprogramma moet ondubbelzinnig zijn: het moet voor de compiler mogelijk zijn om precies één subprogramma te identificeren. De compiler gebruikt de typen en de volgorde van de actuele parameters, de namen van de formele parameters (als de parameters benoemd worden bij de aanroep) en in het geval van functies het type van het resultaat om een overladen subprogramma-aanroep te herkennen. De onderstaande aanroepen van ZET zijn ondubbelzinnig:

```
ZET(LISTING => TRUE);
ZET(BLAUW, MIJN_BUFFER);
```

Maar onderstaande aanroep is wel voor meer dan één uitleg vatbaar:

```
ZET(1372);
```

Het overladen van subprogrammanamen moet met zorg gebeuren: de leesbaarheid van programma's kan er door afnemen. Overladen heeft in feite alleen zin als een equivalente operatie op objecten van verschillende typen moet worden toegepast.

Subprogramma bodies

Een subprogrammadeclaratie is volledig als zowel het subprogramma interface (de wijze van communicatie met de buitenwereld) is gedefinieerd, als ook de 'body' of romp van het subprogramma, waarin de toe te passen algoritme is uitgewerkt. De subprogrammadeclaratie wordt daarom aangevuld met een blok met lokale (alleen binnen het subprogramma geldende) declaraties en met een rij instructies, waarvan een blok wordt gemaakt door ze in te sluiten tussen de gereserveerde woorden **begin** en **end**. In de subprogramma body kan ook zijn beschreven wat moet gebeuren bij het optreden van fouten of bijzondere omstandigheden. Deze 'exception handler' wordt verder in hoofdstuk 17 besproken. Subprogramma's vormen een directe ondersteuning van het principe van beperkt toegankelijk houden van gegevens ('information hiding'), omdat de interne details

van het subprogramma voor de gebruiker 'onzichtbaar' kunnen worden gemaakt. Deze details zijn voor de gebruiker dan ook niet interessant: voor de gebruiker is slechts van belang dat het subprogramma doet wat hij ervan verwacht.

We geven nu een paar voorbeelden van volledige subprogrammadeclaraties en we maken daarbij gebruik van eenvoudige instructies. Enkele nog niet behandelde aspecten komen in het volgende hoofdstuk aan bod, maar de hoofdzaken zullen nu al duidelijk zijn. Bekijk eens de volgende typen declaraties:

```
type LIJST is array (INTEGER range <>) of INTEGER;
type BUFFER(TOTAAL_AANTAL_ELEMENTEN : INTEGER) is
  record
    INDEX   : INTEGER;
    WAARDE : LIJST(1 .. TOTAAL_AANTAL_ELEMENTEN);
  end record;
```

We kunnen nu de declaratie van het in hoofdstuk 6 (paragraaf 6.3) besproken subprogramma PUSH, bedoeld om de elementen op een STACK te plaatsen, verder uitwerken:

```
procedure PUSH(ELEMENT : in INTEGER; OP : in out BUFFER) is
begin
  OP.INDEX := OP.INDEX + 1;
  OP.WAARDE(OP.INDEX) := ELEMENT;
end PUSH;
```

We definieerden hier ELEMENT als een parameter met modus in. Ada maakt het mogelijk deze abstractie dwingend op te leggen; een poging tot toekennen van een waarde aan een dergelijke parameter wordt dan ook afgestraft met een foutmelding tijdens de compilatie van het programma.

We kunnen ook subprogramma's, waarin lokale objecten voorkomen, declareren:

```
with TEXT_IO;
function LEES_VLAG return BOOLEAN is
  GEBRUIKERSINVOER : STRING(1 .. 80);
  begin
    loop
      TEXT_IO.GET(GEBRUIKERSINVOER);
      if GEBRUIKERSINVOER(1 .. 4) = "JA  " then
        return TRUE;
      elsif GEBRUIKERSINVOER(1 .. 4) = "NEE " then
        return FALSE;
      else
        TEXT_IO.PUT_LINE("ANTWOORD MET JA OF NEE");
      end if;
    end loop;
  end LEES_VLAG;
```

We gebruiken in dit subprogramma de **return** instructie om het subprogramma te verlaten en een waarde af te geven, maar misschien is het u al opgevallen, dat er meerdere uitgangen zijn, terwijl toch voor een goede programmeerstijl de regel 'één ingang, één uitgang' geldt. Er zou als volgt van een zogenaamde 'switch' of schakelvariabele gebruik gemaakt kunnen worden:

```
with TEXT_IO;
function LEES_VLAG return BOOLEAN is
  RESULTAAT             : BOOLEAN;
  GEBRUIKERSINVOER      : STRING(1 .. 80);
  CORRECT_ANTWOORD : BOOLEAN := FALSE;
begin
  while not  CORRECT_ANTWOORD do
  loop
    TEXT_IO.GET(GEBRUIKERSINVOER);
    if GEBRUIKERSINVOER(1 ..4 ) = "JA " then
     RESULTAAT := TRUE;
     CORRECT_ANTWOORD := TRUE;
    elsif GEBRUIKERSINVOER(1 .. 4) = "NEE " then
     RESULTAAT := FALSE;
     CORRECT_ANTWOORD := TRUE;
    else
     TEXT_IO.PUT_LINE("ANTWOORD MET JA OF NEE");
    end if;
  end loop;
  return RESULTAAT;
end LEES_VLAG;
```

Toch is deze versie omslachtiger en minder goed leesbaar dan de eerste versie. De stelregel 'eenvoud is het kenmerk van het ware' is meestal nog belangrijker dan bepaalde regels voor een correcte programmastructuur.

We gebruikten hier eenvoudige objectdeclaraties voor lokale objecten (RESULTAAT, GEBRUIKERSINVOER en CORRECT_ANTWOORD), maar ieder item kan lokaal worden gedeclareerd: typen, subtypen, getallen, pakketten, taken en zelfs andere subprogramma's. In hoofdstuk 20 worden deze mogelijkheden nader bekeken.

Als u de voorbeelden goed bestudeert dan zal u nog een subtiel verschil met eerdere voorbeelden opvallen: het gebruik van het gereserveerde woordje **is** in plaats van een puntkomma aan het begin van de body van het subprogramma. Subprogramma bodies zijn op zichzelf staande eenheden, bestaande uit een interface gedeelte (dat overeen moet komen met de eventueel eerdere specificatie van het subprogramma), en een uitwerking van het algoritme. Een subprogrammabody mag overal in het programma voorkomen, waar ook een eenvoudige objectdeclaratie kan staan. (De enige uitzondering is dat een body niet in een pakketspecificatie mag staan, terwijl een subprogrammasepcificatie daar wel mag staan.) Een subprogramma body kan op zichzelf staan in een pakket body, maar kan ook dienen

als aanvulling op het voor de gebruiker toegankelijke gedeelte van het pakket.

Als twee subprogramma's elkaar aanroepen, moeten in ieder geval afzonderlijke subprogrammaspecificaties worden gebruikt. Er is dan sprake van een 'voorwaartse' declaratie, zoals in dit voorbeeld:

```
procedure EERSTE(EEN_PARAMETER : in out MIJN_TYPE) is
   . . .
end EERSTE;
procedure TWEEDE(NOG_EEN_PARAMETER : in out UW_TYPE) is
   . . .
end TWEEDE;
```

Op deze manier gedeclareerd kan TWEEDE wel EERSTE aanroepen, maar omgekeerd kan EERSTE niet TWEEDE aanroepen, omdat een grootheid in Ada pas gebruikt kan worden als hij is gedeclareerd. Door specificatie en declaratie te scheiden wordt wederzijds aanroepen wel mogelijk:

```
procedure EERSTE(EEN_PARAMETER : in out MIJN_TYPE);
procedure TWEEDE(NOG_EEN_PARAMETER : in out UW_TYPE);
   -- nu kunnen eventueel nog declaraties van andere
   -- grootheden volgen
procedure EERSTE(EEN_PARAMETER : in out MIJN_TYPE) is
   . . .
end EERSTE;
procedure TWEEDE(NOG_EEN_PARAMETER : in out UW_TYPE) is
   . . .
end TWEEDE;
```

Nu zijn de interfaces van EERSTE en TWEEDE met de buitenwereld bekend op het moment van uitwerking van hun bodies en kunnen beide subprogramma's elkaar aanroepen. Van deze mogelijkheid kan ook gebruik worden gemaakt om subprogrammaspecificaties bij elkaar te plaatsen in de programmatekst en hun bodies afzonderlijk uit te werken, dit ter vergroting van de leesbaarheid van het geheel. Eigenlijk behoeft in het laatste voorbeeld de specificatie van EERSTE niet uitgeschreven te worden, omdat TWEEDE de procedure EERSTE, die in de programmatekst eerder wordt genoemd, wel automatisch kan zien. Terwille van begrijpelijkheid en consistentie is het aan te raden in een dergelijk geval toch beide specificaties te vermelden.

Als een programmeur er de voorkeur aan geeft declaraties wat meer te verspreiden, dan kan van subeenheden gebruik worden gemaakt. Een voorbeeld:

```
procedure TREK(STEEKPROEF : in out STEEKPROEF_BUFFER)
                                        is separate;
```

De specificatie wordt hier wel gegeven, maar de body wordt geheel

afzonderlijk gecompileerd. In hoofdstuk 20 zullen we laten zien hoe top-down ontwerp door het gebruiken van deze mogelijkheid kan worden vereenvoudigd.

10.2 Subprogramma-Aanroepen



Een subprogramma wordt geactiveerd na een aanroep. Procedure-aanroepen kunnen in een programma als afzonderlijke instructies voorkomen; functie-aanroepen moeten een onderdeel zijn van een expressie, omdat zij immers een waarde retourneren. Het gebruik van subprogramma-aanroepen verhoogt de leesbaarheid van de programmatekst; acties op een hoog abstractieniveau kunnen op die manier benoemd worden en betekenisvolle namen voor subprogramma's zijn daarom van belang. Voorbeelden van specificaties:

```
procedure DOORZOEK_FILE(SLEUTEL   : in NAAM;
                        INDEX     : out FILE_INDEX);
procedure WACHT        (TIJD      : in DUUR := 10.0);
procedure SORTEER      (DATA      : in out NAMEN;
                        VOLGORDE  : in RICHTING := STIJGEND);
procedure SORTEER      (DATA      : in out GETALLEN;
                        VOLGORDE  : in RICHTING := STIJGEND);
procedure ONTSTEEK     (LICHT     : in LOKATIE);
```

In deze voorbeelden hebben we SORTEER overladen, maar van dubbelzinnigheid is geen sprake omdat immers verschillende typen parameters worden gebruikt.

Subprogramma's kunnen op drie manieren worden aangeroepen. Ten eerste via een positionele notatie voor de parameters, zoals dit in veel andere programmeertalen ook gebeurt. In deze positionele notatie moet de volgorde van de actuele parameters overeenkomen met die van de formele:

```
DOORZOEK_FILE("JANSEN, J", RECORD_NUMMER);
WACHT(12.0);
SORTEER(MEDEWERKERSNAMEN,DALEND);
SORTEER(BEOORDELINGSCIJFERS,STIJGEND);
ONTSTEEK(KANTOORLICHTEN);
```

Komt het statische type van de actuele parameter niet overeen met het type van de formele parameter, dan wordt dit tijdens de compilatie in een foutmelding gesignaleerd. Tijdens de verwerking wordt de melding CONSTRAINT_ERROR gegeven als een waarde van een actuele parameter niet valt binnen de aangegeven grenzen van de bijbehorende formele parameter.

Bij de tweede manier van aanroepen worden de parameters benoemd ter vergroting van de leesbaarheid:

```
DOORZOEK_FILE(SLEUTEL=>"JANSEN, J", INDEX=>RECORD_NUMMER);
WACHT(TIJD => 120.0);
SORTEER(DATA => MEDEWERKERSNAMEN, VOLGORDE => DALEND);
```

Op deze wijze documenteren de subprogramma-aanroepen als het ware zichzelf. Benoemde en positionele notatie kunnen in dezelfde aanroep worden gebruikt, maar zodra een parameter wordt benoemd, moeten ook alle daarna volgende parameters benoemd worden. Bij het gebruik van benoemde parameters kan zelfs van de volgorde van de formele parameters worden afgeweken, maar dit wordt niet aangeraden.

De derde manier van aanroepen maakt gebruik van default ('verstek') parameters. De niet gespecificeerde parameters krijgen dan automatisch hun default-waarden. SORTEER kan bijvoorbeeld als volgt worden aangeroepen:

```
SORTEER(MEDEWERKERSNAMEN);
SORTEER(BEOORDELINGSCIJFERS);
```

In beide gevallen wordt de default-volgorde STIJGEND aangenomen. Ook deze mogelijkheid van de taal moet met de nodige voorzichtigheid worden toegepast. Als een subprogramma parameters heeft, die vrijwel nooit van waarde veranderen, dan is gebruik van default-parameters zinvol, maar de leesbaarheid van de programmatekst neemt er door af.

Ook benoemde parameters en default-parameters mogen tesamen worden gebruikt:

```
SORTEER(DATA => MEDEWERKERSNAMEN);
```

We riepen hier een overladen subprogramma aan, maar de Ada compiler kan de ogenschijnlijke dubbelzinnigheid oplossen via het type van de actuele parameter.

Functie-aanroepen zien er iets anders uit dan procedure-aanroepen:

```
function COS(HOEK : in RADIALEN) return FLOAT;
function TEMPERATUUR(VOELER : in VOELERNAAM) return FLOAT;
function "+" (X,Y : in MATRIX) return MATRIX;
```

Deze subprogramma's kunnen nu bijvoorbeeld als volgt worden aangeroepen:

```
AFSTAND := LENGTE * COS(30.0);
WAARDE  := TEMPERATUUR(VOELER => VLEUGELTIP);
SOM     := "+" (EERSTE_MATRIX,TWEEDE_MATRIX);
```

Zowel benoemde als positionele parameters zijn toegelaten, evenals default-parameters. In het laatste voorbeeld werd een prefixnotatie gebruikt om de "+" functie te activeren, maar ook de gebruikelijke infixnotatie is toegelaten, als de operatorspecificatie tenminste direct zichtbaar is:

```
SOM := EERSTE_MATRIX + TWEEDE_MATRIX;
```

We riepen hier de overladen operator "+" aan, maar het is voor de compiler mogelijk op grond van de typen van de operanden vast te stellen, dat het hier gaat om een matrixoptelling.

Ook functies zonder parameters zijn mogelijk. Vooral voor predicaten (logische beweringen) is deze vorm geschikt:

```
function IS_TOEGELATEN_OPERATIE return BOOLEAN;
function BEVOEGD_TOT_TOEGANG    return BOOLEAN;
```

Dergelijke functies worden aangeroepen zonder actueel parameter gedeelte:

```
if BEVOEGD_TOT_TOEGANG then ...
```

Merk op dat een parameterloze functie-aanroep niet te onderscheiden is van een eenvoudige objectnaam.

10.3 Toepassingen Voor Ada Subprogramma's



We zagen dus, dat subprogramma's uitvoerbare programma-eenheden zijn, die algoritmen bevatten. Subprogramma's zijn meer dan in de programmatekst bij elkaar geplaatste instructies; hun toepassingsgebied is drieledig:

- Hoofdprogramma-eenheden.
- Definitie van functies.
- Definitie van operaties op abstracte datatypen.

We geven nu voorbeelden van elk van deze drie toepassingen.

Subprogramma's als hoofdprogramma's

Ada kent geen afzonderlijke constructie voor hoofdprogramma's. Elk subprogramma dient onderdeel te vormen van een Ada systeem, dat vanuit de programmeeromgeving kan worden geactiveerd. Op welke wijze de hoofdeenheid wordt aangeroepen, wordt niet door de taal

gedefinieerd, maar wordt aan de programmeeromgeving overgelaten. Een bepaalde implementatie van de taal zou wellicht een pragma (een compileraanwijzing) MAIN kunnen voorschrijven, maar het invoeren van een hoofdprogramma op deze wijze is eigenlijk in strijd met Ada's opzet. Ook verbiedt de taal niet, dat een 'hoofd' subprogramma met de programmeeromgeving communiceert via parameters.

Het is een goede gewoonte grote systemen over een 'hoofd'programma te laten beschikken, dat volstrekt implementatie-onafhankelijk is. Zoals al in hoofdstuk 4 werd benadrukt, betekent een modulaire opbouw, dat modulen op het hoogste abstractieniveau aangeven *wat* een systeem moet doen en dat lagere modulen aangeven *hoe* dit gedaan moet worden. Een subprogramma op het hoogste abstractieniveau moet daarom alleen de essentiële systeemtypen en objecten bevatten, tesamen met de subprogramma's, pakketten of taken die de bijbehorende operaties mogelijk maken.

Een permanent draaiend systeem, bijvoorbeeld, dat geregeld gegevens administreert, kan op het hoogste niveau als volgt worden beschreven:

```
with RECORDER,VOELER;
procedure ADMINISTREER_LIJN_TOESTAND is
  VOLTAGE : VOELER.VOLTAGE_TYPE;
begin
  loop
    VOELER.LIJN(METING => VOLTAGE);
    RECORDER.ZEND(VOLTAGE);
  end loop;
end ADMINISTREER_LIJN_TOESTAND;
```

Net zoals bij onze eerste twee ontwerpproblemen worden hier via de **with** clausule, bibliotheekeenheden ingevoerd (RECORDER en VOELER). Het instrumentarium, dat bij deze eenheden behoort, komt op die manier beschikbaar, maar de implementatie-details blijven verborgen. Deze wijze van inkapselen maakt het mogelijk ook grote en complexe systemen te blijven beheersen.

Definitie van functies

Bij top-down functioneel ontwerp begint men de probleemanalyse op het hoogste niveau van abstractie en ontbindt men vervolgens de probleemoplossing in steeds elementairdere functies. Dit proces verloopt analoog aan de door ons geïntroduceerde objectgerichte ontwerpmethode.

Zodra de functies zijn vastgesteld kunnen subprogramma's worden geformuleerd, die deze functies uitvoeren. Als het resultaat van een functie één enkele waarde is, dan ligt het voor de hand een functie-subprogramma te gebruiken. In andere gevallen dient het procedure-subprogramma te worden toegepast. Bij functies behoren acties en het is aan te raden dit in subprogrammanamen tot uitdrukking

te laten komen. Actieve werkwoordsvormen verdienen hier de voorkeur. Procedures kunnen namen hebben als: TREK_LANDINGSGESTEL_IN of CONTROLEER_GRENZEN. Functies kunnen namen hebben die de aard van de geretourneerde waarde weergeven: COS, RANDOM of MEETWAARDE. Als de geretourneerde waarde een BOOLEAN is, dan is een eigenschapsomschrijvende naam zinvol: PROCES_IS_BEEINDIGD, LANDINGSGESTEL_IS_UIT of WAARDEN_ZIJN_TOEGELATEN.

Op elk niveau moeten steeds alleen die subprogramma's zichtbaar zijn, die essentieel zijn voor het huidige stadium van de oplossing. Op lagere niveaus kunnen natuurlijk meer subprogramma's nodig zijn om operaties van een hoger niveau mogelijk te maken, maar deze dienen verborgen te blijven. Een eenvoudig programma voor gegevensverwerking kan er bijvoorbeeld zo uitzien:

```
with STEEKPROEF;
procedure ANALYSEER_MEETWAARDEN is
  GEMETEN_WAARDEN     : STEEKPROEF.WAARDEN;
  AANGEPASTE_WAARDEN : STEEKPROEF.WAARDEN;
  procedure TREK_STEEKPROEF(DATA : out STEEKPROEF.WAARDEN) is separate;
  procedure TEST_GRENZEN(DATA : in out STEEKPROEF.WAARDEN) is separate;
  procedure PAS_CURVE_AAN   (DATA : in STEEKPROEF.WAARDEN;
                        AANPASSING: out STEEKPROEF.WAARDEN) is separate;
  procedure RAPPORTEER      (DATA : in STEEKPROEF.WAARDEN) is separate;
begin
  TREK_STEEKPROEF (GEMETEN_WAARDEN);
  TEST_GRENZEN    (GEMETEN_WAARDEN);
  PAS_CURVE_AAN   (GEMETEN_WAARDEN, AANPASSING => AANGEPASTE_WAARDEN);
  RAPPORTEER      (AANGEPASTE_WAARDEN);
end ANALYSEER_MEETWAARDEN;
```

In de subprogrammaspecificaties wordt weer de **separate** clausule gebruikt; de subprogrammanamen zelf zijn zichtbaar, maar hun bodies worden geheel afzonderlijk gecompileerd en zijn daarom zowel wat betreft hun programmatekst als wat betreft hun logica voor de gebruiker onzichtbaar.

Door op deze manier subprogramma's te gebruiken wordt de leesbaarheid en daardoor de onderhoudbaarheid van de programmatuur verbeterd. Wil men deze techniek doeltreffend gebruiken, dan moeten de operaties op een weloverwogen manier worden onderverdeeld. In een real-time omgeving echter, kan een dergelijke onderverdeling in een groot aantal subprogramma's leiden tot een te grote vertraging, wegens de benodigde extra administratie ('overhead'). De pragma (compileraanwijzing) INLINE kan worden gebruikt om deze overhead te elimineren, terwijl de voordelen van modulaire opbouw met behulp van subprogramma's niet verloren gaan. Het gevolg van de aanwijzing INLINE is, dat elke subprogramma-aanroep vervangen wordt door de subprogramma body tijdens de compilatie. Hierbij worden de actuele parameters gesubstitueerd, zodat noch parameteroverdracht, noch subroutine adres administratie noodzakelijk is. De gebruiker blijft de desbetreffende subprogramma's echter

zien als logische op zichzelf staande eenheden. Het pragma INLINE moet vermeld worden in hetzelfde gedeelte als waar de subprogramma's worden gedeclareerd en dit gebeurt als volgt:

```
pragma INLINE(TREK_STEEKPROEF,TEST_GRENZEN,
              PAS_CURVE_AAN,RAPPORTEER);
```

Argumenten bij het pragma zijn de subprogrammanamen. De betekenis van een subprogramma verandert in geen enkel opzicht door het gebruik van het pragma.

Definitie van operaties op abstracte datatypen

Een type wordt gekarakteriseerd door een waardenverzameling en een verzameling operaties van toepassing op objecten van dit type (zie hoofdstuk 8). Primitieve typen (bijvoorbeeld INTEGER) kennen impliciete operaties (optellen, vermenigvuldigen), maar vaak heeft de programmeur behoefte aan het definiëren van eigen (abstracte) typen met bijbehorende operaties. In hoofdstuk 13 zullen we nader op de details ingaan, maar al in onze eerste twee ontwerpproblemen hebben we gezien hoe hiertoe subprogramma's kunnen worden gebruikt. Pakketten worden gebruikt om de logische abstracties van de gebruiker in te kapselen en een correct gebruik dwingend op te leggen. Een voorbeeld:

```
package VERBONDEN_LIJST is
  type ELEMENT is limited private;
  procedure VOEG_TOE (AAN   : in out ELEMENT;
                      DATA  : in     ELEMENT);
  procedure VERWIJDER(UIT   : in out ELEMENT;
                      DATA  : out    ELEMENT);
  procedure IS_LEEG   (DATA : in out ELEMENT) return BOOLEAN;
private
  . . .
end VERBONDEN_LIJST;
```

Deze pakketspecificatie bevat alleen het specificatiegedeelte van de subprogramma's. De bijbehorende subprogramma bodies worden in de pakket body opgenomen. Met bovenstaand voorbeeld hebben we een abstract datatype gecreëerd, VERBONDEN_LIJST.ELEMENT genaamd, waarop drie en niet meer dan drie operaties zijn toegelaten: VOEG_TOE, VERWIJDER en IS_LEEG. De uitwerking van de subprogramma's blijft op dit niveau verborgen.

In dit hoofdstuk hebben we laten zien hoe subprogramma's kunnen worden gebruikt om algoritmen in te kapselen. Vanzelfsprekend behoeven deze acties op het hoogste abstractieniveau een verdere uitwerking op lagere niveaus. In het volgende hoofdstuk zullen we Ada's instructies op het elementairste niveau gaan bestuderen.

Oefeningen

1. Schrijf een subprogrammaspecificatie voor de operatie NORMALISEER. Deze operatie dient de waarde van een getal van het type FLOAT op de een of andere manier te veranderen (hoe doet hier niet ter zake).

2. Schrijf een subprogrammaspecificatie voor de operatie MATRIX_VERMENIGVULDIGING, die twee matrices vermenigvuldigt. Herschrijf vervolgens deze operatie als een functie, met gebruikmaking van het symbool '*'.

3. Schrijf de specificatie voor een subprogramma BESTEL_DINER, met als parameters de gangen van de maaltijd. Neem default waarden op voor alle parameters.

4. Schrijf de declaratie voor een BOOLEAN functie IS_LEEG, die gebruikt wordt om stack underflow te testen. De enige benodigde parameter is de te testen stack.

11 EXPRESSIES EN INSTRUCTIES

Als we de productiviteit van programmeurs meten in aantal regels correcte code per dag, dan blijkt dit aantal betrekkelijk constant te zijn en onafhankelijk van het niveau van de gebruikte programmeertaal. Zowel voor assembleertalen als voor hogere programmeertalen is de productie ongeveer 10 regels code per dag [1]. Omdat een instructie in een hogere programmeertaal krachtiger is dan een assembler instructie, neemt de productiviteit duidelijk toe naarmate het niveau van de programmeertaal toeneemt. In hogere programmeertalen, waarin van subprogramma's gebruik kan worden gemaakt is de netto productiviteit het hoogst: door de gebruiker geformuleerde algoritmen kunnen daarin als primitieve operaties worden toegepast.

Op een zeker moment moeten dergelijke algoritmen uitgewerkt worden en deze uitwerking moet liefst zo helder en duidelijk mogelijk zijn. Op dezelfde manier als waarop abstracte datatypen worden geconstrueerd uit meer elementaire typen, worden subprogramma's geconstrueerd met behulp van elementaire instructies, die ons de structuren verschaffen voor de besturing van de algoritmen en de hulpmiddelen voor het berekenen van waarden. In dit hoofdstuk zullen we expressies en instructies ('statements') in Ada nader bestuderen.

11.1 Namen



Voordat we iets met een object kunnen doen, moeten we in staat zijn naar dat object te verwijzen via een naam. In Ada heeft een naam altijd betrekking op een gedeclareerde grootheid, zoals een object, een getal, een type, subtype, subprogramma, pakket, taak, taakingang of exceptie. In de volgende declaraties zijn de grootheden in hoofdletters toegelaten namen in Ada:

```
type PROCES_TYPE is (ACTIEF,GEREED,GEBLOKKEERD,BEEINDIGD);
type TEL          is array (PROCES_TYPE) of
                     INTEGER range 0 .. INTEGER'LAST;
type TEL_NAAM     is access TEL;
ROOSTER_TABEL     : TEL;
LOKAAL_ROOSTER    : TEL_NAAM;
PROCES_TOESTAND: PROCES_TYPE;
subtype COEFFICIENT is FLOAT digits 7 range -1.0 .. 1.0;
subtype MAAT      is INTEGER range 1 .. 4;
type MATRIX       is array(MAAT,MAAT) of COEFFICIENT;
MATRIX_1,MATRIX_2: MATRIX;

type KLEP         is record
                     NAAM              : STRING(1 .. 10);
                     PLAATS            : STRING(1 .. 10);
                     OPEN              : BOOLEAN;
                     STROOM_SNELHEID : FLOAT;
                    end record;
subtype AANTAL_KLEPPEN is INTEGER range 1 .. 100;
type KLEPPEN      is array(AANTAL KLEPPEN) of KLEP;
KLEP_INDEX        : AANTAL_KLEPPEN;
KLEP_RECORD       : KLEPPEN;

PI                : constant := 3.141_592_65;

IS_LEEG,IS_ACTIEF: BOOLEAN;
VOLTAGE_1,VOLTAGE_2 : FLOAT;
TEL_1,TEL_2       : AANTAL_KLEPPEN;
type RADIALEN     is new FLOAT;
function COS(HOEK : in RADIALEN) return FLOAT;
```

In het vervolg van dit hoofdstuk zullen we deze declaraties gebruiken in voorbeelden van expressies en instructies.

Voor het verwijzen naar een grootheid in zijn geheel kunnen we eenvoudig de naam van die grootheid gebruiken (bijvoorbeeld MATRIX_1 of KLEP_INDEX), maar om naar een gedeelte van een samengestelde grootheid te verwijzen, is een andere notatie nodig. Voor array-elementen en onderdelen van een verzameling taken (zie hoofdstuk 16) gebruiken we de indexnotatie. Gebruikmakend van de eerder gegeven declaraties, kunnen we bijvoorbeeld als volgt naar elementen verwijzen:

```
MATRIX_1(1,4)
ROOSTER_TABEL(PROCES_TYPE'SUCC(GEBLOKKEERD))
KLEP_RECORD(37)
```

Uit het ROOSTER_TABEL voorbeeld blijkt dat een index ook een expressie mag zijn, in dit geval zelfs met een attribuut. Valt de waarde van de index buiten het interval van toegelaten indexwaarden, dan wordt tijdens de verwerking de melding CONSTRAINT_ERROR gegeven.

We kunnen niet alleen verwijzen naar afzonderlijke componenten, maar ook naar een rij opeenvolgende componenten, een schijf of *slice* genaamd. Voorbeelden hiervan:

```
ROOSTER_TABEL(GEREED .. BEEINDIGD)   --  3 componenten
KLEP_RECORD(1 .. 50)                 -- 50 componenten
```

Ook hier ontstaat de toestand CONSTRAINT_ERROR als de waarde van een index buiten de begrenzingen valt.

Slices komen vooral goed van pas, als grote gegevensblokken van het ene naar het andere array moeten worden gekopieerd. Dit gebeurt bijvoorbeeld in de volgende waardetoekenningsopdracht:

```
KLEP_RECORD(1 .. 20) := KLEP_RECORD(21 .. 40);
```

De twintig waarden vanaf KLEPPEN(21) worden toegekend aan de eerste twintig elementen. Omdat in Ada het linker en rechter lid van een toekenningsopdracht afzonderlijk worden geëvalueerd, is een overlapping van de slices ook toegestaan:

```
KLEP_RECORD(1 .. 10) := KLEP_RECORD(6 .. 15);
```

Deze instructie heeft het volgende effect: eerst worden de huidige waarden van de elementen 6 tot en met 15 geëvalueerd en vervolgens worden deze waarden aan de elementen 1 tot en met 10 toegekend.

Bij records en access-waarden kan men naar delen van een grootheid verwijzen via de componentselectie- of puntnotatie. Deze puntnotatie kan ook worden gebruikt bij subprogramma's, paketten, taken en blokken. Voorbeelden:

```
KLEP_RECORD(7).PLAATS
KLEP_RECORD(KLEP_INDEX).STROOM_SNELHEID
LOKAAL_ROOSTER(ACTIEF)
LOKAAL_ROOSTER.all
```

Bij de eerste twee voorbeelden is sprake van combinatie van indicering en puntnotatie. In het derde voorbeeld wordt via een index naar een component van het object verwezen en in het vierde voorbeeld verwijst **all** naar het gehele object, waartoe via de access-waarde toegang wordt verkregen.

Attributen van gedeclareerde grootheden kunnen in de meeste gevallen via de accentnotatie worden benaderd. Attributen kunnen behoren bij objecten, typen, subtypen, subprogramma's en taken. Een volledige lijst van voorgedefinieerde attributen is opgenomen als Appendix D. Eigenschappen van de eerder gedeclareerde grootheden kunnen bijvoorbeeld als volgt worden benaderd:

```
PROCES_TYPE'FIRST          -- de waarde ACTIEF
IS_LEEG'ADDRESS            -- het fysieke adres van het object
VOLTAGE_1'SIZE             -- het aantal bits in het object
ROOSTER_TABEL'RANGE        -- een korte notatie voor ROOSTER-
                              _TABEL'FIRST .. ROOSTER-
                              _TABEL'LAST
```

Attributen kunnen, zoals de meeste andere benoemde grootheden, binnen expressies worden gebruikt.

11.2 Waarden



Om met behulp van expressies nieuwe waarden voor objecten te kunnen genereren moeten we om te beginnen een mogelijkheid hebben elementaire waarden aan te geven. In Ada kunnen zowel scalairen als samengestelde waarden worden weergegeven. Eenvoudige scalaire waarden kunnen via een letterlijke representatie worden aangegeven:

```
1_024              -- een geheeltallige numerieke waarde
0.398_892_138      -- een reële numerieke waarde
GEBLOKKEERD        -- een waarde uit een opsomming of enumeratie
"OPSLAGRUIMTE"     -- een string
null               -- de lege access-waarde, verwijst naar niets
'b'                -- een schriftteken
16#FFE#            -- een hexadecimaal (zestientallig) getal
```

Letterlijk weergegeven numerieke waarden (literals) hebben het type 'universal_integer' of 'universal_real' en worden intern letterlijk voorgesteld. Als deze waarden worden toegekend aan een object of als ze worden gebruikt in een expressie om een nieuwe waarde te berekenen, dan wordt zo'n literal geconverteerd naar een getalsvoorstelling van het juiste type, afhankelijk van de context. Als bijvoorbeeld het object MATRIX_1(1,1), dat gedeclareerd werd met zeven significante cijfers, de waarde 0.398_892_138 krijgt toegekend, dan wordt deze letterlijke voorstellingswijze geconverteerd naar een getal met minstens zeven significante cijfers. Dit kan de programmeur van nut zijn: wat hem of haar betreft is de precisie van alle getalsrepresentaties onbeperkt. We bevelen daarom dan ook aan, numerieke constanten via deze letterlijke getalsrepresentatie te declareren en niet te beperken tot het een of andere numerieke type.

Tot nu toe werkten we met enkelvoudige namen, maar in Ada kunnen ook samengestelde waarden worden gebruikt voor arrays en records. Deze waarden worden *aggregaten* genoemd en kunnen worden weergegeven via de positionele of benoemde componentennotatie.

Voor het ééndimensionale array-object ROOSTER_TABEL kunnen we bijvoorbeeld waarden genereren door de componenten op te sommen:

```
(7,3,1,0)                       -- positionele notatie
(ACTIEF      => 7,              -- dezelfde waarden, maar nu benoemd
 GEREED      => 3,
 GEBLOKKEERD => 1,
 BEËINDIGD   => 0)
(ACTIEF .. BEEINDIGD => 0       -- gebruik van een interval
(ACTIEF | BEEINDIGD => 0,       -- gebruik van een keuze-alternatief
          others    => 1)
```

Er zijn dus heel wat varianten mogelijk; de benoemde componentennotatie is in de meeste gevallen te verkiezen, want deze is de meest leesbare. Elke indexwaarde kan worden benoemd, zoals in het tweede voorbeeld, of we kunnen een interval gebruiken, zoals in het derde voorbeeld is gebeurd. Deze laatste mogelijkheid is vooral nuttig als verscheidene componenten dezelfde waarde hebben. De ACTIEF|BEEINDIGD constructie betekent dat beide componenten de waarde nul krijgen; **others** verwijst vervolgens naar alle componenten die nog geen waarde toegewezen kregen. Als een **others** clausule wordt gebruikt in een aggregaat, moet het de laatst vermelde keuze zijn (het kan ook de enig vermelde keuze zijn). Als de **others** clausule wordt gebruikt, dan moet het uit de context duidelijk zijn wat de begrenzingen van de geaggregeerde grootheid zijn.

Van belang is te weten, dat bij het gebruik van de benoemde componentennotatie, de geaggregeerde expressie voor elke index afzonderlijk wordt uitgewerkt. Om dit te verduidelijken een voorbeeld:

```
(1 .. 5 => new TEL)
```

Er worden hier vijf objecten van het type TEL gegenereerd; het is dus niet zo dat alle vijf componenten verwijzen naar hetzelfde object TEL.

Mocht een aggregaat (en in feite geldt dit voor iedere expressie) voor meer dan één uitleg vatbaar zijn, dan kan de programmeur de keuze van waarden expliciet maken via een zogenaamde *gekwalificeerde expressie*. Deze expressie heeft dezelfde vorm als een attribuut, met gebruik van het accentteken, gevolgd door de expressie (die weer een aggregaat kan zijn). Stel we hebben de volgende declaraties:

```
type WIJZERSTAND is array (1 .. 5) of FLOAT digits  7;
type POLYNOOM    is array (1 .. 7) of FLOAT digits 15;
```

Beschouw nu de waarde:

```
(1 .. 3 => 10.1, others => 0.0)
```

De compiler zal deze waarde als een dubbelzinnig aggregaat herkennen, want het kan zowel van het type WIJZERSTAND als van het type POLYNOOM zijn. Deze dubbelzinnigheid kan worden geëlimineerd via een gekwalificeerde expressie:

```
POLYNOOM'(1 .. 3 => 10.1, others => 0.0)
```

De kwalificatie geeft dus expliciet het type van de expressie aan, en dit is vooral gewenst in het geval van een **others** clausule.

Als de elementen van een aggregaat niet overeenkomen met de begrenzingen van de componenten, dan komt het systeem net als bij conflicten tussen actuele en formele parameters in de toestand CONSTRAINT_ERROR. Elke geaggregeerde waarde moet verder nog compleet zijn; dat wil zeggen, er moet voor zijn gezorgd dat alle componenten een waarde krijgen toegewezen.

In het geval van *n*-dimensionale arrays moet het aggregaat worden opgedeeld in ééndimensionale componenten. De volgende aggregaten zijn bijvoorbeeld toegelaten voor een object van het type MATRIX (een 4 bij 4 matrix van het type COEFFICIENT):

```
MATRIX'((0.0,0.0,0.0,0.0),     -- op nul zetten van de matrix
        (0.0,0.0,0.0,0.0),
        (0.0,0.0,0.0,0.0),
        (0.0,0.0,0.0,0.0))
MATRIX'(1 .. 4 =>              -- een eenvoudiger manier
        (0.0,0.0,0.0,0.0))
MATRIX'(1 .. 4 =>              -- nog een manier
        (1 .. 4 => 0.0))
```

In ons eerste voorbeeld werd het aggregaat geformuleerd als een verzameling ééndimensionale aggregaten. In het tweede en derde voorbeeld hebben we het bereik van de rij-index van de matrix benoemd; elk van de aldus benoemde rijen krijgen vervolgens de waarde van een ééndimensionaal aggregaat toegekend. In het geval van multidimensionale arrays worden de aggregaten geschreven in volgorde van de indices.

Aggregaten voor records worden geschreven in dezelfde notatie als voor arrays, maar als de **others** clausule wordt gebruikt, dan moet naar minstens één element direct verwezen worden. Alle elementen, waarnaar via **others** wordt verwezen moeten verder van hetzelfde type zijn. Voor objecten van het type KLEP kunnen we bijvoorbeeld de volgende geaggregeerde waarden creëren:

```
KLEP'("WATER     ","LOODS     ",TRUE,37.65) -- positionele notatie
KLEP'(NAAM            => "WATER     ",      -- benoemde notatie
      PLAATS          => "LOODS     ",
      OPEN            => TRUE
      STROOM_SNELHEID => 37.65)
```

Deze geaggregeerde vormen kunnen natuurlijk weer worden gecombineerd. Om bijvoorbeeld een beginwaarde te geven aan het KLEP_RECORD object zou het volgende aggregaat kunnen worden benoemd:

```
KLEPPEN'(1 .. 100)          =>
        (NAAM               => "          ",
         LOKATIE            => "          ",
         OPEN               => FALSE,
         STROOM_SNELHEID    => 0.0))
```

Hier worden 100 verschijningen van het geïnitialiseerde record-aggregaat genoemd. In dit geval kan de **others** clausule worden gebruikt:

```
KLEPPEN'(1 .. 100           =>
        (OPEN               => FALSE,
         STROOM_SNELHEID    => 0.0,
         others             => "          "))
```

Omdat Ada behalve in het geval van access-typen niet automatisch beginwaarden toekent aan objecten, kunnen aggregaten handig worden gebruikt om defaultwaarden toe te kennen via een samengestelde objectdeclaratie.

11.3 Expressies



Nu we objecten kunnen benoemen en primitieve waarden kunnen toekennen zullen we onze aandacht richten op de expressie met behulp waarvan nieuwe waarden kunnen worden berekend. Natuurlijk kunnen niet met ieder stel grootheden waarden worden berekend: het vermenigvuldigen van twee subprogrammanamen is bijvoorbeeld weinig zinvol. De operanden binnen expressies mogen alleen zogenaamde *primairen* zijn, zoals:

```
"AANWIJZING"          -- een stringwaarde
10.125                -- een numerieke waarde
(7,3,1,0)             -- een geaggregeerde waarde
MATRIX_1              -- een naam
new TEL'(0,0,0,0)     -- een verwijzing
COS(37.5)             -- een functie-aanroep
INTEGER(123.9)        -- een typeconversie
COEFFICIENT'(0.53)    -- een gekwalificeerde expressie
(3*4)                 -- een expressie tussen haakjes
null                  -- een lege waarde
```

Elke primair heeft een waarde en een type. Operaties zijn alleen toegelaten tussen primairen met hetzelfde type. Wel is, zoals we in hoofdstuk 8 zagen, een expliciete typeconversie toegelaten voor afgeleide typen en alle numerieke typen. Fouten wegens conflicten tussen statische typen kunnen meestal al tijdens de compilatie worden ontdekt. Als tijdens de uitwerking van een expressie blijkt dat de berekende waarde buiten de capaciteit van de gebruikte computer valt, dan komt het systeem in de toestand NUMERIC_ERROR:

```
GEMIDDELDE := TOTAAL/SOM;  -- NUMERIC_ERROR als SOM = 0.0
```

Als een expressie een waarde voortbrengt die buiten de begrenzingen van het bijbehorende object valt, dan volgt de toestand CONSTRAINT_ERROR, zodra de waardetoekenning aan het object wordt uitgevoerd:

```
type BEPERKT_BEREIK is range 0 .. 10;
INDEXJE : BEPERKT_BEREIK;
INDEXJE := 100;  -- CONSTRAINT_ERROR op moment van toekenning
```

Een goede compiler zou zelfs al een waarschuwing kunnen geven tijdens de compilatie.

We hebben nu de toegelaten operanden in Ada expressies beschreven. De taal beschrijft ook zes klassen operatoren, die op deze operanden kunnen worden toegepast. In volgorde van afnemende prioriteit zijn deze operatoren:

```
**    not  abs              -- operatoren met hoogste prioriteit
*   /  mod  rem             -- vermenigvuldigingsoperatoren
+   -                       -- unaire optelling
+   -   &                   -- binaire optelling
=   /=  <   <=  >  >=       -- relationele operatoren
and  or  xor                -- logische operatoren
```

Deze operatoren werken met de precisie van de basistypen. Als we bijvoorbeeld twee objecten van het type COEFFICIENT (een subtype van FLOAT) met elkaar vermenigvuldigen, dan worden de berekeningen uitgevoerd met de precisie van COEFFICIENT'BASE, dus met de precisie van FLOAT.

Alle operatorsymbolen, met uitzondering van '/=', kunnen met behulp van een functiedeclaratie overladen worden, zoals in hoofdstuk 10 werd beschreven. Een voorbeeld:

```
function "/"(X,Y : in COMPLEX.GETAL) return COMPLEX.GETAL;
```

We hebben nu de '/' operator overladen. Bij het gebruik van de operator kan zowel de infix- als de prefixnotatie worden gebruikt:

```
RESULTAAT := "/"(EERSTE,TWEEDE);  -- prefixnotatie
RESULTAAT := EERSTE/TWEEDE;       -- infixnotatie
```

Bij het gebruik van de infixnotatie worden de aanhalingstekens niet gebruikt. De prioriteiten van operatoren veranderen door het overladen niet.

Behalve de bovengenoemde symbolen heeft Ada ook zogenaamde *kortgesloten* logische operatoren (**and then** en **or else**), waarbij de tweede operand alleen geëvalueerd wordt als de eerste respectievelijk waar en niet waar is. Deze *kortgesloten* logische operatoren hebben dezelfde prioriteit als de standaard logische operatoren. Tests of een element al dan niet tot een verzameling behoort (**in** en **not in**) hebben dezelfde prioriteit als de relationele operatoren. Hier komen wij later in dit hoofdstuk op terug. De voorgedefinieerde functie **abs** tenslotte, is toepasbaar op ieder numeriek type en geeft de absolute waarde van een expressie.

Operatoren met de hoogste prioriteit worden het eerste toegepast. Als operatoren gelijke prioriteit bezitten, dan is de uitwerking van links naar rechts, of in een andere volgorde die tot hetzelfde resultaat leidt. Haakjes veranderen de volgorde van uitwerking natuurlijk:

```
(1 >= 9) and (2 <= 10)   -- resultaat is FALSE
1.5**(-3)                -- resultaat is 0.296_296_296...
2.5/0.5 + 2.0            -- resultaat is 7.0
2.5/(0.5 + 2.0)          -- resultaat is 1.0
```

Een optimaliserende compiler kan mogelijkerwijs de volgorde van verwerking van operaties van gelijke prioriteit wijzigen. Dat wil zeggen dat elk programma, dat impliciet van een bepaalde volgorde van verwerking uitgaat, *onjuist* is. We bedoelen hier met *onjuist*, dat het mogelijk is dat de Ada compiler geen fout ontdekt en dit betekent weer, dat de uitkomst bij het gebruik van verschillende implementaties onvoorspelbaar is. Het niet mogen uitgaan van een bepaalde volgorde lijkt misschien een nogal beperkende regel, maar deze is bedoeld ter bevordering van de overdraagbaarheid en betrouwbaarheid. Trouwens, ook voor de leesbaarheid is het altijd beter om haakjes te gebruiken om de volgorde van de bewerkingen aan te geven, in plaats van van de lezer te verwachten dat hij de impliciete prioriteitsregels zal uitpuzzelen. Vergelijk bijvoorbeeld eens:

```
1.5*X**2 + 6.5*X + 4.7
(1.5*(X**2)) + (6.5*X) + 4.7
```

Deze twee expressies leiden tot hetzelfde resultaat, maar het gebruik van haakjes maakt de tweede expressie een stuk leesbaarder.

Ada is een 'sterk getypeerde' taal (alle typen moeten expliciet worden gedeclareerd) en daarom zijn de operatoren die in expressies kunnen worden gebruikt maar voor bepaalde typen voorgedefinieerd. Dit is in tabel 11-1 aangegeven (zie ook het pakket STANDARD in Appendix C). De meeste operatoren worden in Ada

Tabel 11-1: Voorgedefinieerde Operatoren in Ada

Operator	Operatie	Type Operanden	Type Resultaat
**	machtsverheffen	L: integer R: integer >= 0	integer
		L: float R: integer	float
*	vermenigvuldigen	integer	integer
		float	float
		L: fixed R: integer	fixed
		L: integer R: fixed	integer
		L: fixed R: fixed	universal_fixed
/	deling	integer	integer
		float	float
		L: fixed R: integer	fixed
		L: fixed R: fixed	universal_fixed
mod	modulus	integer	integer
rem	remainder (rest)	integer	integer
+	unaire plus	numeriek type	numeriek type
-	unaire min	numeriek type	numeriek type
abs	absolute waarde	numeriek type	numeriek type
not	unaire ontkenning	BOOLEAN	BOOLEAN
		array van BOOLEANs	idem
+	optelling	numeriek type	mumeriek type
-	aftrekken	numeriek type	numeriek type
&	concatenatie	ééndimensionale arrays	idem
		array en component	array
		component en array	array
		component en component	array
=	gelijkheid	elk type	BOOLEAN
/=	ongelijkheid	elk type	BOOLEAN
<	kleiner dan	elke scalair	BOOLEAN
		array	BOOLEAN
<=	kleiner of gelijk	elke scalair	BOOLEAN
		array	BOOLEAN
>	groter dan	elke scalair	BOOLEAN
		array	BOOLEAN
>=	groter of gelijk	elke scalair	BOOLEAN
		array	BOOLEAN
in	element van	L: elke scalair R: range	BOOLEAN
	verzameling	L: elk type R: subtype	BOOLEAN
not in	geen element van	L: elke scalair R: range	BOOLEAN
	verzameling	L: elk type R: subtype	BOOLEAN
and	conjunctie	BOOLEAN	BOOLEAN
		array van BOOLEANs	array
and then	conjunctie	BOOLEAN	BOOLEAN
	(kortgesloten)	array van BOOLEANs	array
or	inclusieve	BOOLEAN	BOOLEAN
	disjunctie	array van BOOLEANs	array
or else	inclusieve	BOOLEAN	BOOLEAN
	disjunctie	array van BOOLEANs	array
	(kortgesloten)		
xor	exclusieve	BOOLEAN	BOOLEAN
	disjunctie	array van BOOLEANs	array

op precies dezelfde manier toegepast als in andere hogere programmeertalen. In alle gevallen hebben de operatoren de gebruikelijke wiskundige betekenis.

De operator voor machtsverheffing (**) is alleen voorgedefinieerd voor gehele machten. Het team dat de taal ontwierp heeft machtsverheffing tot een niet gehele macht bewust niet opgenomen; zij meenden dat deze operatie binnen het probleemgebied van de 'ingebedde' systemen weinig zou worden toegepast. Een efficiënt algoritme voor machtsverheffing is trouwens sterk machine-afhankelijk. Als machtsverheffing tot een niet-gehele macht nodig is, dan moet de programmeur zelf deze functie schrijven en de machtsverheffingsoperator daarmee overladen. Beter is nog gebruik te maken van een exponentiële functie met de gewenste precisie uit een mathematisch pakket. Dit zijn voorbeelden van correct gebruik van machtsverheffing:

```
3 ** 7           -- resultaat is 2187
2.718 ** 3       -- resultaat is 20.079_290_...
3.142 ** (-1)    -- resultaat is 0.318_268_...
```

Een geheeltallige waarde kan niet direct tot een negatieve macht worden verheven, omdat dan het resultaat immers niet geheeltallig zou zijn. Pogingen daartoe tijdens de verwerking leiden tot de foutmelding CONSTRAINT_ERROR.

Operaties op gehele getallen met de operaties *, /, **mod**, **rem** leiden tot exacte resultaten; dezelfde operaties uitgevoerd op reële getallen leiden tot afrondingsfouten. In hoofdstuk 8 gaven wij aan dat Ada zorgt dat een bepaald numeriek type aan zekere minimale eisen van precisie blijft voldoen; de inexacte resultaten worden dan ook afgerond op een bepaalde dichtstbijzijnde waarde. Ga eens uit van de onderstaande declaraties:

```
type FIXED is delta 0.0001 range -1_000.0 .. +1_000.0;
FIXED_1    : FIXED := 0.1;
FIXED_2    : FIXED := 0.1;
INTEGER_1  : INTEGER := 2;
INTEGER_2  : INTEGER := 3;
FLOAT_1    : FLOAT  := 1.0;
FLOAT_2    : FLOAT  := 2.0;
```

Dit zijn toegelaten expressies met vermenigvuldigingsoperaties:

```
FIXED_1 * FIXED_2        -- resultaat 0.01
INTEGER_1 / INTEGER_2    -- resultaat is 1 (type INTEGER)
FLOAT_1 / FLOAT_2        -- resultaat is afronding op 0.5
                            (type FLOAT)
```

De **rem** operator geeft als resultaat de rest na uitvoering van een geheeltallige deling. Het teken is gelijk aan dat van de teller. De **mod** of modulus-operatie geeft bij gelijk teken van teller en noemer eveneens de rest na geheeltallige deling; bij verschillend

teken wordt de rest berekend door een geheel aantal malen de noemer van de teller af te trekken, zodat de teller juist overschreden wordt. Dit leidt bijvoorbeeld tot de volgende resultaten:

L	R	L rem R	L mod R
12	5	2	2
14	5	4	4
12	-5	2	-3
14	-5	4	-1
-12	5	-2	3
-14	5	-4	1
-12	-5	-2	-2
-14	-5	-4	-4

De mod operatie wordt het meest gebruikt. (Een rekenregel, waarbij bijvoorbeeld 12 mod -5 gelijk is aan 2, is echter gebruikelijker.)

De unaire operatoren (+, -, not, abs) worden op één enkele operand toegepast en leiden tot een resultaat van hetzelfde type als dat van de operand. De identiteit (+) en de negatie (-) kunnen worden toegepast op elk numeriek type. De operator not is ook toegelaten op een array van BOOLEANs. Dit lijkt op het eerste gezicht vreemd, maar dit geeft een mogelijkheid tot het uitvoeren van bitsgewijze operaties. We zullen in hoofdstuk 17 laten zien, dat BOOLEAN arrays naar een bitvector kunnen worden afgebeeld en daarom kunnen operaties op arrays van BOOLEANs worden gezien als bitmanipulaties.

Voor de binaire operatoren (+, -, &) zijn twee operanden nodig. De bewerkingen worden uitgevoerd volgens de normale rekenkundige regels. De concatenatie-operator (&) kan worden toegepast op elk type ééndimensionaal array, maar is in hoofdzaak bedoeld voor stringoperaties. (*Concatenatie* is het samenvoegen van twee arrays tot één array met als lengte de som van de lengtes van de componenten.) Deze operator is standaard zodanig overladen, dat ook het concateneren van een array en één enkel element mogelijk is:

```
"FOUT_MELDING" & CR & LF   -- concatenatie van een string met
                              Carriage Return en Line Feed
"A" & "BCDEFG"             -- concatenatie van twee strings
Z + 0.1                    -- Z moet een reëel type hebben
```

De relationele operatoren (=, /=, <, >, <=, >=) zijn bedoeld voor tests op gelijkheid, ongelijkheid en orderelaties. Tests voor gelijkheid en ongelijkheid kunnen op elk datatype worden toegepast dat niet 'limited private' is. De orderelatie is van toepassing op elk scalair type en elk array type. De operatoren in en not in testen of een element al dan niet tot een verzameling behoort. Hun rechterlid kan een interval zijn (voor een test op vallen binnen bepaalde grenzen) of een typebeschrijving (voor een test op voldoen aan een bepaalde randvoorwaarde). Voorbeelden:

```
J not in 1 .. 10       -- als J binnen het interval is resultaat FALSE
"AA" > "B"             -- resultaat is FALSE
X in COEFFICIENT       -- controle op subtype
```

De logische operatoren (and, or, xor) bewerkstelligen de gebruikelijke Boole'se operaties. De logische operatoren zijn, evenals de not operatie, gedefinieerd voor zowel BOOLEAN waarden, als voor arrays van BOOLEAN elementen. Op deze manier is het mogelijk bitmanipulaties direct in Ada uit te voeren. Deze elementaire logische operatoren hebben alle dezelfde prioriteit: binnen expressies moet de gewenste prioriteit dus met behulp van haakjes worden aangegeven.

In bepaalde gevallen kan de zogenaamde 'kortgesloten' versie van de operatoren van pas komen bij de evaluatie van samengestelde BOOLEAN expressies. Als and then wordt gebruikt, dan wordt het rechterlid alleen dan geëvalueerd als het linkerlid bij evaluatie TRUE bleek te zijn. Wordt daarentegen or else gebruikt, dan wordt het rechterlid alleen dan bepaald als het linkerlid FALSE was. Voorbeelden:

```
IS_ACTIEF or IS_KLAAR          -- een eenvoudige BOOLEAN operatie
Y /= 0.0 and then X/Y > 5.0    -- een kortgesloten operatie (rechter-
                                  lid wordt alleen bepaald als linker-
                                  lid TRUE is)
IS_LEEG xor IS_ACTIEF          -- FALSE als linker- en rechterlid
                                  dezelfde waarde hebben
```

Zouden we in het tweede voorbeeld niet de kortgesloten versie van de operator hebben gebruikt, dan bestaat de mogelijkheid van een fout tijdens de verwerking indien Y de waarde nul heeft. De expressie zou wel anders zijn te formuleren (if X /= 0.0 then if X/Y > 5.0), maar vaak is de kortgesloten versie korter en duidelijker.

We behandelen tot besluit het begrip statische expressie. Een *statische expressie* is een expressie die niet afhangt van tijdens de verwerking van het programma berekende waarden; er mogen dus geen variabelen of dynamische attributen in voorkomen. Een expressie is statisch als zijn termen alle bestaan uit letterlijk genoteerde waarden ('literals') en expressies (ook letterlijke enumeraties), constanten die via statische expressies werden geinitialiseerd, statische attributen en als statisch gekwalificeerde expressies. Voorbeelden van statische expressies:

```
3                  -- een numerieke literal
INTEGER'FIRST      -- een statisch attribuut
abs(PI**3)         -- een statische expressie
```

Bij bestudering van de syntaxdiagrammen in Appendix A zult u zien dat de taal op verscheidene plaatsen het gebruik van statische expressies vereist.

Op grond van de nu beschreven operatoren in Ada, zijn dit voorbeelden van toegelaten expressies:

```
ROOSTER_TABEL(BEEINDIGD)
KLEP_RECORD(KLEP_INDEX).STROOMSNELHEID not in 1.0 .. 5.0
(PI ** (TEL_1 rem 4)) >= MATRIX_1(1,1)
IS_ACTIEF and then ROOSTER_TABEL(BEËINDIGD) = 0
(MATRIX_1(1,1) *MATRIX_1(2,2)) - (MATRIX_1(3,3) *(MATRIX_1(4,4))
(VOLTAGE_1 - VOLTAGE_2) / 2.781
```

We kunnen nu met gegeven waarden nieuwe waarden berekenen en het wordt nu tijd de besturing van algoritmen met behulp van instructies te bekijken.

11.4 Instructies



Het uitvoeren van een instructie heeft een actie tot gevolg. Net zoals abstracte datatypen worden geconstrueerd met behulp van elementairder typen, zo zijn instructies de elementaire bouwstenen voor de specificatie van acties. Het is mogelijk abstracte en samengestelde acties te definiëren door middel van subprogramma's en vervolgens deze acties weer als elementaire bouwstenen voor een hoger abstractieniveau te gebruiken. Instructies worden ook gebruikt voor het beïnvloeden van de volgorde van uitvoering. C. Bohm en G. Jacopini [2] toonden aan dat drie besturingsstructuren voldoende zijn voor het formuleren van algoritmen, en wel:

- de sequentiële structuur
- de voorwaardelijke of conditionele structuur
- de herhalings- of iteratieve structuur

Voor elk van deze structuren heeft Ada één ingang en (meestal) één uitgang. In de volgende paragrafen zullen we de Ada-instructies behandelen als varianten op de hier genoemde basisstructuren.

Sequentiële besturing

Onder sequentiële besturing worden de instructies na elkaar in leesvolgorde uitgevoerd. Dit geldt in Ada voor de volgende categorieën instructies:

- waardetoekenning
- de 'null' instructie

- procedure-aanroep
- 'return' instructie
- de blockspecificatie

Ook behandelen we hier de beruchte goto-instructie, hoewel deze wel degelijk een afwijking van de sequentiële verwerking veroorzaakt.

Alle instructies in Ada worden afgesloten (dus niet gescheiden) door een puntkomma. Hierdoor wordt het voor compilers gemakkelijker, zich te herstellen van kettingreacties van fouten en ook wordt de leesbaarheid door deze vaste regel vergroot. Het is niet verboden meer dan één instructie op een regel te plaatsen, maar beter is het om dit niet te doen. We zullen in deze paragraaf nog meer aanwijzigingen geven voor het weergeven van instructies; zie voor een samenvatting Appendix B, *Stijlvol programmeren in Ada.*

Een *waardetoekenningsopdracht* vervangt de huidige waarde van een variabele door een nieuwe waarde, die volgt uit de evaluatie van een expressie. De instructie heeft als linkerlid de variabele, waaraan een waarde wordt toegekend en als rechterlid de expressie, waaruit deze waarde volgt. Beide leden wordt gescheiden door het samengestelde teken ':='. Linker- en rechterlid moeten hetzelfde type hebben: in het geval van statische typen wordt het zondigen tegen deze regel al tijdens de compilatie ontdekt. Valt het resultaat van een expressie buiten de grenzen van de door de machine voorstelbare waarden, dan wordt tijdens de verwerking de foutmelding NUMERIC_ERROR gegeven. Valt de berekende waarde buiten de voor de variabele vastgestelde grenzen, dan volgt de melding CONSTRAINT_ERROR.

Dit zijn voorbeelden van toegelaten waardetoekenningsopdrachten:

```
VOLTAGE_1                   := VOLTAGE_2 + 24.0;
MATRIX_1                    := MATRIX_2;
LOKAAL_ROOSTER.all          := TEL'(0,0,0,0);
ROOSTER_TABEL(GEREED)       := ROOSTER_TABEL(GEREED ) + 1;
KLEP_RECORD(TEL_1).OPEN     := TRUE;
KLEP_RECORD(1 .. 10)        := KLEPPEN'(1 .. 10   =>
                                  (NAAM    => "RESERVE  ",
                                   PLAATS  => "LOODS    ",
                                   OPEN    => FALSE,
                                   STROOMSNELHEID => 0.0));
```

Merk op, hoe in de laatste instructie van een 'slice' of plak gebruik werd gemaakt, om in één keer een rij elementen een waarde toe te kennen.

Bij het ontwerp van Ada is men van de filosofie uitgegaan, dat alle opdrachten tot actie expliciet moeten worden geformuleerd, daarom bestaat er ook een expliciete opdracht om 'niets' te doen: de null-instructie. De null-instructie is te vergelijken met de opmerking 'deze pagina is met opzet blanco gehouden' in een rapport. De pagina is dan weliswaar niet meer blanco, maar het is zo aan de lezer duidelijk, dat er geen tekst per abuis is vergeten. Het effect van de

null-instructie is slechts, dat verder wordt gegaan met de volgende instructie. Ook bij de compilatie kan de instructie worden genegeerd, dus heeft de instructie geen vertragende invloed bij de verwerking. Het enige doel is het vergroten van de programmatekst. De instructie ziet er zo uit:

```
null;
```

Later zullen we zien dat deze instructie wordt gebruikt in case-instructies, om keuzen die mogen worden genegeerd toch expliciet te vermelden, verder in een recorddeclaratie om een leeg variant gedeelte aan te geven en in de body van een nog niet verder uitgewerkt subprogramma (een zogenaamde 'stub').

De volgende zuiver sequentiële instructie is de *procedure-aanroep*. Hoe deze er uitziet hebben we al in hoofdstuk 10 laten zien, dus de syntax herhalen we hier niet. Wel komen we hier terug op het belang van de procedure als uitdrukkingsmiddel, maar de gebruiker kan de procedure zien als een elementaire actie, benoemd door de procedurenaam.

De return-instructie staat direct met het gebruik van subprogramma's in verband. Deze instructie beëindigt de uitvoering van een procedure of een functie. Een functie moet minstens één en mag meerdere return-instructies bevatten. Binnen een procedure ziet de instructie er zo uit:

```
return;
```

Indien deze instructie wordt uitgevoerd, dan wordt verwerking van de procedure beëindigd en wordt de besturing teruggegeven aan het aanroepende programma. Alle waarden van out of in out parameters worden daarbij meegegeven: dit laatste gebeurt niet als het subprogramma niet op normale wijze wordt beëindigd. In het geval van functies gebeurt hetzelfde, maar tevens wordt via de functienaam een waarde geretourneerd. Bekijk eens het volgende voorbeeld:

```
function IS_ONEVEN(WAARDE : in INTEGER) return BOOLEAN is
begin
  if (WAARDE rem 2) = 0 then
    return FALSE;
  else
    return TRUE;
end IS_ONEVEN;
```

Als return wordt gevolgd door een expressie dan moet het type van de geretourneerde waarde overeenkomen met het type gespecificeerd in de functiedeclaratie, anders volgt ofwel een syntax-error, ofwel een CONSTRAINT_ERROR tijdens de verwerking. Indien het einde van de functie-body wordt bereikt zonder dat sprake was van een return-instructie, dan volgt eveneens een CONSTRAINT_ERROR.

Als in het bovenstaande voorbeeld WAARDE gelijk was aan 2, dan zou (WAARDE **rem** 2) = 0 de waarde TRUE opleveren en IS_ONEVEN zou FALSE retourneren. Het **else**-gedeelte zou in dit geval niet worden uitgevoerd.

In het algemeen is het overzichtelijker als een subprogramma maar één **return**-instructie bevat, maar soms is het duidelijker als van deze regel wordt afgeweken. Een subprogramma kan altijd zo herschreven worden, dat het maar één **return** bevat, door het toevoegen van een extra variabele (zie hoofdstuk 10), maar soms wordt het geheel er daardoor niet duidelijker op. In dit geval kunnen we handig gebruik maken van het feit dat de geretourneerde waarde een BOOLEAN is:

```
function IS_ONEVEN(WAARDE : in INTEGER) return BOOLEAN is
begin
  return((WAARDE rem 2) /= 0);
end IS_ONEVEN;
```

De laatste zuiver sequentiële besturingsstructuur is het *blok*. De blok-instructie maakt het mogelijk een rij instructies, tesamen met lokale declaraties en desgewenst instructies die aangeven wat te doen in uitzonderingsgevallen, ('exception handler') tot één geheel te maken. De benodigde geheugenruimte voor gegevens die lokaal zijn ten opzichte van een blok, behoeft pas te worden toegekend bij het binnenkomen van dat blok. Van buiten het blok zijn gegevens, die lokaal zijn ten opzichte van dit blok, niet te bereiken.

Een blok en een subprogramma lijken op het eerste gezicht veel op elkaar. Het essentiële verschil is dat een procedure door willekeurig veel andere programma-eenheden kan worden aangeroepen, het bereik of de *scope* van een blok is echter beperkt tot het stuk programmatekst waar het blok in voorkomt. De *scope* van een eenheid is het gebied van de programmatekst, waarbinnen deze eenheid bekend is en bestaat. Als we bijvoorbeeld een object declareren in een subprogramma, dan is de scope van dat object het punt waar het voor het eerst wordt genoemd, tot aan het eind van het subprogrammabody. Hetzelfde geldt voor binnen een blok gedeclareerde grootheden. Hun scope begint bij het punt waar zij voor het eerst worden genoemd en eindigt aan het einde van het blok. Een blok kan daarom niet worden aangeroepen; het wordt slechts sequentieel uitgevoerd.

Het blok wordt vaak gebruikt voor het creëren van een lokale foutbehandeling, of voor het declareren van lokale objecten of typen. Een voorbeeld:

```
WISSEL;
  declare
    TEMP : FLOAT;
  begin
    TEMP      := VOLTAGE_1;
    VOLTAGE_1 := VOLTAGE_2;
    VOLTAGE_2 := TEMP;
  end WISSEL;
```

Een blok wordt aangegeven met behulp van de gereserveerde woorden **begin** en **end**. Voor de leesbaarheid kan een bloknaam, zoals WISSEL worden toegevoegd. Als blokken binnen blokken voorkomen (als zij 'genest' zijn), dan is het verstandig om ze namen te geven: dit biedt de mogelijkheid om toch naar lokale objecten te verwijzen (zie hoofdstuk 20). Opnieuw een voorbeeld:

```
BUITEN:
  declare
    TEMP : FLOAT;        --+
  begin                    |   Scope van BUITEN.TEMP
    ...                    |
    BINNEN:                |   A
    declare                |
      TEMP : FLOAT;  --+   |
    begin              |   |   Scope van BINNEN.TEMP
      ...              |   |   B
    end BINNEN;      --+   |
  ...                      |   C
  end BUITEN;          ----+
```

Bij de punten A en C is alleen de declaratie van BUITEN.TEMP zichtbaar. We bedoelen met *zichtbaar*, dat het object een unieke naam heeft, waarmee het benaderd kan worden. Bij punt B zijn zowel BUITEN.TEMP als BINNEN.TEMP zichtbaar. De regel in Ada is nu als volgt: als een declaratie binnen een blok dezelfde naam gebruikt als in het buitenste blok, dan zal de compiler steeds uitgaan van de lokale declaratie. Zouden wij dus op punt B van de naam TEMP gebruik maken, dan wordt automatisch aangenomen dat het gaat om BINNEN.TEMP. Door de puntnotatie te gebruiken is echter op punt B ook de variabele TEMP uit het buitenblok bereikbaar via BUITEN.TEMP, terwijl BINNEN.TEMP of eenvoudig TEMP verwijst naar de variabele TEMP uit het binnenblok.

Het hierboven gegeven voorbeeld is eigenlijk nogal kunstmatig: het gebruik van dergelijke geneste blokstructuren is nu niet direct aanbevelenswaardig. Toch kan een programmeur die meewerkt aan een groot project niet altijd zeker weten welke variabele-namen door anderen worden gebruikt en de hierboven beschreven scope-regels geven in ieder geval volledige controle over de bereikbaarheid en zichtbaarheid van objecten. (Zie verder hoofdstuk 20.)

Tenslotte bespreken we de **goto**-instructie, hoewel deze niet zuiver sequentieel is. Toch lijkt dit ons de geschikste plaats om deze instructie te behandelen. We zouden boekdelen kunnen vullen over verhitte discussies tussen voor- en tegenstanders van het gebruik van **goto**. Hier volstaan we met te stellen, dat we in Ada waarschijnlijk wel zonder **goto** zouden kunnen, maar indien men een **goto**-instructie wil gebruiken, dan moet deze een beperkte reikwijdte hebben en goed gedocumenteerd zijn.

In Ada is de scope van **goto** zodanig beperkt, dat springen in een samengestelde instructie (een **if**, **loop**, **accept**, **case**

of een blok) niet is toegelaten, Ook mag niet gesprongen worden van de ene if, case of select-instructie in de andere, en ook niet in subprogramma-, taak- of pakketbodies. De goto-instructie ziet er zo uit:

```
goto SLUITEN;
```

De goto-instructie bestaat uit het gereserveerde woord goto gevolgd door de naam van een label. Elke instructie kan vooraf worden gegaan door een label, dat wordt aangegeven door de tekens '<<' en '>>'. Bijvoorbeeld:

```
<<SLUITEN>>START_SLUIT_PROCEDURE;
```

Voorwaardelijke besturing

Voorwaardelijke besturing selecteert één bepaalde rij instructies uit een aantal mogelijke rijen. Deze selectie wordt uitgevoerd op basis van een of ander criterium of op basis van de waarde van een expressie. Er zijn in Ada twee instructies voor voorwaardelijke besturing beschikbaar, namelijk:

- if
- case

De if-instructie selecteert uit een aantal mogelijke rijen één of geen rij instructies, afhankelijk van de logische waarde van één of meer expressies. Deze expressies zijn van het type BOOLEAN, dat wil zeggen zij hebben de waarde TRUE of de waarde FALSE. De if-instructie kan op drie manieren worden geformuleerd:

```
if TEL_1 < 5 then      -- eenvoudige if-then constructie
  TEL_1 := 9;
end if;

if KLEP_RECORD(1).OPEN then    -- if-then-else constructie
  KLEP_RECORD(2).OPEN := TRUE;
  KLEP_RECORD(3).OPEN := FALSE;
else
  KLEP_RECORD(2).OPEN := FALSE;
  KLEP_RECORD(3).OPEN := TRUE;
end if;

if VOLTAGE_1 > VOLTAGE_2 then  -- geneste if constructie
  VOLTAGE_1 := VOLTAGE_2;
elsif VOLTAGE_1 < VOLTAGE_2 then
  VOLTAGE_2 := VOLTAGE_1;
else
  null;
end if;
```

Steeds wordt de if-instructie afgesloten door **end if** en de mogelijke alternatieven worden steeds ingesloten door een paar gereserveerde woorden. Hoogstens één alternatief wordt uitgevoerd. Als in het eerste voorbeeld TEL_1 < 5 waar is, dan wordt de volgende instructie (TEL_1 := 9) uitgevoerd. Als de waarde van de expressie FALSE is, dan wordt de besturing onmiddellijk aan het einde van de if-instructie gegeven. Ditzelfde geldt in het tweede voorbeeld, zij het dat daar het **else**-gedeelte wordt uitgevoerd als de voorwaarde FALSE is. In het derde geval zijn er drie, elkaar uitsluitende mogelijkheden, dus precies één van de instructie-rijen zal worden gekozen. Vooral als de voorwaarden erg ingewikkeld zijn, is het verstandig ook een **null**-gedeelte toe te voegen.

Ook met de **case**-structuur kan geselecteerd worden uit alternatieven. Met behulp van de **case**-structuur kan echter gekozen worden uit een groot aantal alternatieven, op grond van de waarde van een discrete expressie. (Een *discrete expressie* is een expressie met een geheeltallige waarde, of een waarde uit een enumeratie.) Voorbeelden:

```
case PROCES_TOESTAND is
  when ACTIEF       => ROOSTER_TABEL(ACTIEF) := 1;
                       IS_ACTIEF := TRUE;
  when GEREED       => ROOSTER_TABEL(GEREED) :=
                          ROOSTER_TABEL(GEREED) + 1;
                       IS_ACTIEF := FALSE;
  when GEBLOKKEERD  => ROOSTER_TABEL(GEBLOKKEERD) :=
                          ROOSTER_TABEL(GEBLOKKEERD) + 1;
                       IS_ACTIEF := FALSE;
  when BEEINDIGD    => ROOSTER_TABEL BEEINDIGD) :=
                          ROOSTER_TABEL(BEEINDIGD) + 1;
                       IS_ACTIEF := FALSE;
end case;

case TEL_1 is
  when 1       => KLEP_RECORD(TEL_1).OPEN := TRUE;
  when 2|3     => KLEP_RECORD(TEL_1).OPEN := FALSE;
  when 5 .. 10 => KLEP_RECORD(TEL_1).OPEN := FALSE;
  when others  => KLEP_RECORD(TEL_1).OPEN := TRUE;
                  KLEP_RECORD(TEL_1).STROOMSNELHEID := 1.0;
end case;
```

In beide voorbeelden wordt precies één alternatief gekozen, afhankelijk van de waarde van de expressie na het woord **case**. Deze expressie wordt de *case selector* genoemd. De case-instructie moet één en slechts één alternatief hebben voor elke mogelijke waarde van de expressie. Deze alternatieven moeten elkaar uitsluiten en alle mogelijkheden dekken. Meer dan één waarde mag tot dezelfde keuze leiden, zoals in het laatste voorbeeld (2|3) en (5 .. 10). De keuze **others** dekt alle mogelijkheden die niet eerder zijn gespecificeerd en dit moet dus altijd de laatste mogelijkheid zijn (als **others** tenminste gebruikt wordt). Vooral als er sprake is van een groot aantal moge-

lijke waarden, dus bijvoorbeeld bij een selector van het type INTEGER, kan **others** goed van pas komen.

Als algemene aanwijzing kan gelden, dat de **case**-instructie gebruikt wordt wanneer sprake is van voorwaarden, gebaseerd op scalaire waarden. In andere gevallen wordt de **if**-instructie gebruikt, met name in gevallen waarin sprake is van complexe logische expressies.

Iteratieve besturing

De laatste klasse besturingsstructuren maakt het mogelijk een rij instructies nul of meer keren te herhalen. Ada kent drie verschillende iteratieve structuren, alle van de vorm 'één ingang en (meestal) één uitgang':

- de elementaire lus
- de 'for'-lus
- de 'while'-lus

In verband met de lus of **loop** staat de **exit**-instructie, die we ook in deze paragraaf zullen behandelen.

De elementaire lus is de eenvoudigste structuur; het is een nimmer eindigende herhaling:

```
loop
   . . .
end loop;
```

Dit is een lus met één ingang, maar geen uitgang. Deze komt voor als buitenste lus van een taak, die continu moet worden uitgevoerd, zoals bijvoorbeeld het uitvoeren van metingen bij een procesbewaking.

Een lus kan worden verlaten door middel van de **exit**-instructie. De direct na de lus volgende instructie is dan de eerstvolgende die wordt uitgevoerd. Er zijn verschillende vormen voor de **exit**-instructie, onder andere:

```
exit;                         -- verlaat de lus
exit BUITENSTE_LUS;           -- verlaat de benoemde lus
exit when TEMP > MAX_TEMP;    -- verlaat de lus als aan de voor-
                                 waarde is voldaan
exit BUITENSTE_LUS when IS_ACTIEF;
                              -- verlaat de benoemde lus als aan
                                 de voorwaarde voldaan is
```

We zien dus dat de **exit**-instructie een loop-naam en/of een voorwaarde kan bevatten. Wordt de naam weggelaten, dan wordt aangenomen dat het om de lus gaat die de instructie direct omsluit.

Als er sprake is van een aantal geneste lussen, dan kunnen deze benoemd worden en dan is het mogelijk expliciet aan te geven welke lus verlaten moet worden:

```
BUITENSTE_LUS:                                  ┐
  loop                                          │
    . . .                                       │  A
    BINNENSTE_LUS:                              │
      loop                          ┐           │
        . . .                       │ B         │
      end loop BINNENSTE_LUS;───────┘           │
    . . .                                       │  C
  end loop BUITENSTE_LUS;                       ┘
```

De regels voor het verlaten van een lus komen overeen met de eerder gegeven regels voor de scope van variabelen. Als we bij A of C een instructie **exit** of **exit** BUITENSTE_LUS geven, dan gaat de besturing naar het eind van de buitenste lus. Staat bij B de instructie **exit** of **exit** BINNENSTE_LUS, dan wordt de BINNENSTE_LUS verlaten, maar bevinden we ons nog steeds binnen de BUITENSTE_LUS. Plaatsen we echter bij B de instructie **exit** BUITENSTE_LUS, dan wordt ook de BUITENSTE_LUS verlaten. Net als in het geval van de **return**-instructie zijn er ook meervoudige **exit**-instructies mogelijk binnen een lus.

De tweede iteratieve besturingsstructuur is de **for**-lus. Een **for**-lus wordt ook wel een *tellende* lus genoemd, omdat het hier gaat om een aftelbaar aantal herhalingen. Deze structuur wordt opgebouwd met behulp van een elementaire lus, voorafgegaan door een **for**-iteratie clausule. Deze clausule bevat een impliciet gedeclareerde tel-variabele, die met een gespecificeerde stapgrootte het gespecificeerde interval afloopt. De waarde van de tel-variabele mag binnen de lus gebruikt worden, maar kan niet worden veranderd (bijvoorbeeld via een waardetoekenning). De tel-grootheid is lokaal ten opzichte van de lus; erbuiten is hij niet bekend. Een voorbeeld:

```
for INDEX in ACTIEF .. BEEINDIGD
  loop
    ROOSTER_TABEL(INDEX) := 0;
  end loop;

for INDEX in reverse TOTAAL_AANTAL_KLEPPEN
  loop
    KLEP_RECORD(INDEX).OPEN := FALSE;
  en loop;
```

In het eerste voorbeeld doorloopt INDEX de waarden ACTIEF, GEREED, GEBLOKKEERD, en BEEINDIGD, in deze volgorde omdat dit de volgorde is waarin de opsomming gedeclareerd werd. De lus wordt dus vier keer herhaald. In het tweede voorbeeld doorloopt INDEX de waarden 100 tot 1, aflopend; de lus wordt dus honderd keer herhaald.

De begrenzing voor de tel-variabele behoeft niet statisch te zijn:

```
for INDEX in 1 .. TEL_1
  loop
    . . .
  end loop;
```

De waarde van TEL_1 wordt buiten de lus vastgesteld; daarbij is het van belang te weten dat het interval voor INDEX slechts één maal wordt vastgesteld. Dus zelfs als TEL_1 binnen de lus een andere waarde zou krijgen, dan zou toch de lus 10 keer worden herhaald als TEL_1 bij het binnengaan van de lus de waarde 10 had.

Moet de herhaling over het hele waardebereik van een type worden uitgevoerd, dan kan dat op verschillende manieren worden aangegeven:

```
for I in KLEPPEN'RANGE
  loop
    . . .
  end loop;

for I in KLEPPEN'FIRST .. KLEPPEN'LAST
  loop
    . . .
  end loop;
```

Beide voorbeelden hebben hetzelfde effect. Merk nogmaals op dat de variabele I niet hoeft te worden gedeclareerd; de compiler zal het juiste type voor deze tel-variabele vaststellen.

De laatste herhalingsstructuur is de **while**-lus. Bij deze structuur wordt een rij instructies herhaald, zolang aan een bepaalde voorwaarde voldaan is. Deze mogelijkheid is van belang als het aantal herhalingen tevoren niet bekend is, of als de herhaling afhangt van een logische voorwaarde. De structuur wordt gevormd door de elementaire lus te laten voorafgaan door de **while**-constructie:

```
while (ROOSTER_TABEL(1).STROOMSNELHEID > 10.0)
      and (not IS_LEEG)
  loop
    . . .
  end loop;
```

Bovenstaande lus wordt alleen binnengegaan als de stroomsnelheid initieel groter is dan 10 en als de waarde van IS_LEEG gelijk aan FALSE is. De rij instructies binnen de lus wordt nu herhaald zolang de voorwaarde TRUE blijft. Nog een laatste opmerking over de structuur: de voorwaarde wordt steeds getest aan het begin van de lus. Als een test op een ander punt nodig is, dan moet dit met behulp van de **exit**-instructie gebeuren.

We bespraken in dit hoofdstuk enkele van Ada's mogelijkheden voor het berekenen van nieuwe waarden en voor het beschrijven van de verwerkingsvolgorde binnen een algoritme. Nu we over deze nieuwe gereedschappen beschikken kunnen we in het volgende hoofdstuk het database opvraagsysteem, dat we in hoofdstuk 9 begonnen te ontwikkelen, voltooien.

Oefeningen

1. Declareer een driedimensionaal array-object KUBUS met componenten van het type FLOAT en met 10 elementen per zijde. Geef vervolgens de acht hoeken van deze KUBUS aan.

2. Declareer een record object CONDITIE_CODE met de BOOLEAN elementen CARRY, NUL en NEGATIEF. Verder met een component PRIORITEIT met een INTEGER bereik van 0 tot en met 31. Benoem vervolgens de component PRIORITEIT.

3. Declareer een access object, dat verwijst naar CONDITIE_CODE. Benoem vervolgens de verwijzingen naar het access object, het gehele object waarnaar verwezen wordt en de component NUL.

4. Schrijf een aggregaat voor het KUBUS object, zodanig dat één zijde van het object de waarde 1.0 krijgt en alle andere componenten de waarde -1.0.

5. Schrijf een aggregaat voor het CONDITIE_CODE object met de waarde FALSE voor CARRY en NUL, de waarde TRUE voor NEGATIEF en de waarde 7 voor PRIORITEIT.

6. Formuleer een Ada expressie voor de volgende vergelijkingen:
   (a) De oppervlakte van een cirkel.
   (b) Het volume van een bol.
   (c) Het product van een rij en een kolom van twee matrices.

*7. Schrijf een rij instructies voor de volgende acties:
   (a) De berekening van de som van twee matrices.
   (b) Een functie met als inputparameter een NATURAL waarde en als resultaat de waarde die men krijgt door de cijfers van de ingevoerde waarde van achteren naar voren op te schrijven. (Input 12345 geeft dus als resultaat 54321.)
   (c) De berekening van de som van alle elementen van het object KUBUS.
   (d) Het doorzoeken van het object KUBUS en een BOOLEAN variabele GEVONDEN op TRUE zetten als een element met een negatieve waarde wordt gevonden.

(e) Zie (d), maar nu dient het totale aantal negatieve elementen te worden geteld.
(f) Zie (d), maar stop de iteratie zodra een element met waarde 0.0 wordt gevonden.

12 HET TWEEDE ONTWERPPROBLEEM: VERVOLG

Nu we Ada's subprogramma's en instructies hebben behandeld als het gereedschap waarmee acties kunnen worden beschreven, kunnen we ook de oplossing van het probleem dat we in hoofdstuk 9 beschreven verder uitwerken.

12.1 Nogmaals Het Probleem



Het probleem betrof het ontwikkelen van een eenvoudig database benaderingssysteem. Informeel formuleerden we de oplossing als volgt:

> **Een database in APSE wordt gebruikt om informatie op te slaan over elke programma-eenheid van een bepaald project. De over elke eenheid opgeslagen informatie omvat de naam van de eenheid, de programmeur, de voortgangsstatus, identificatie van de huidige versie en een verwijzing naar het bijbehorend specificatiedocument. De interactieve bewerkingen, waartoe de gebruiker opdracht kan geven zijn onder meer: het produceren van een lijst van projectgegevens (gesorteerd op programmeursnaam of op grond van basisspecificaties), het weergeven van alle gegevens over een gekozen programma-eenheid en het verzamelen van kentallen over de ontwikkelingsfase van alle programma-eenheden. Een opvraagsessie duurt totdat de gebruiker een verzoek om de sessie te beëindigen invoert.**

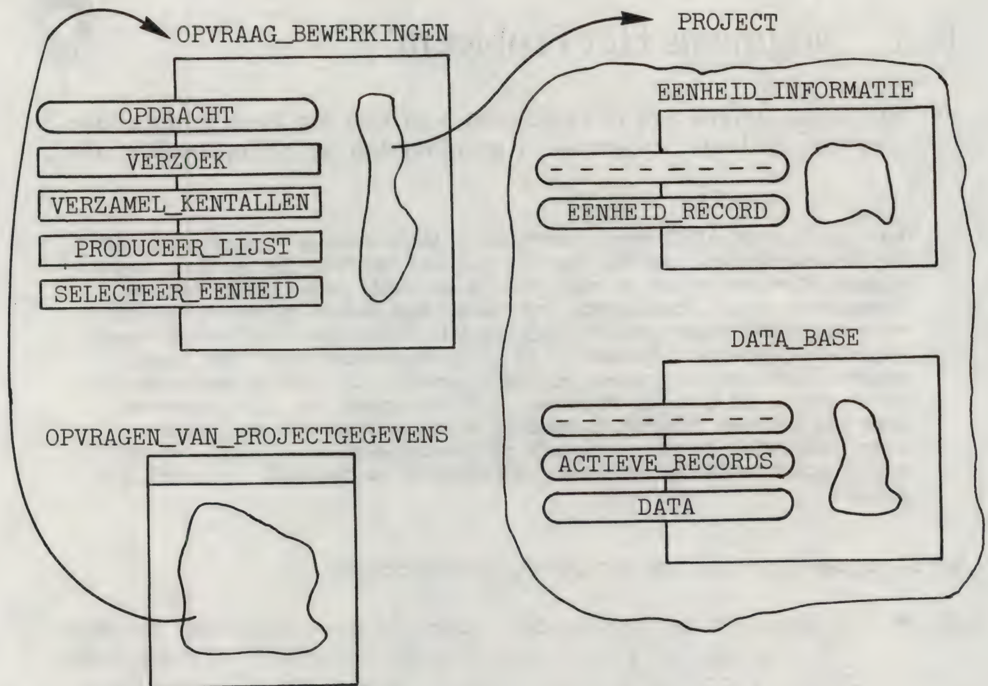
We formuleerden ook de volgende specificaties:

- Om een lijst te produceren voert de gebruiker het sorteercriterium in (de programmanaam of de specificatie-identificatie). Vervolgens wordt een op grond van dit criterium gesorteerde lijst van programma-eenheden gemaakt. Als de gebruiker bijvoorbeeld de *naam* als sorteersleutel gebruikt, dient alle informatie alfabetisch te worden gesorteerd op programmanaam.
- Een bepaalde programma-eenheid wordt door de gebruiker geselecteerd door de naam van de eenheid op te geven.

Vervolgens wordt een lijst met alle gegevens van die eenheid geproduceerd.

- Voor het produceren van kentallen dient de absolute en relatieve frequentie van de staten van ontwikkeling van alle eenheden te worden bepaald (de *status* kan zijn: ontwerpfase, coderingsfase, testfase en operationeel). Er moet dus een lijst, gesorteerd naar status, worden geproduceerd, met bij elke status het aantal programma-eenheden en het aantal als percentage van het totaal.

We gaven ons ontwerp, met gebruikmaking van de objectgeoriënteerde ontwerpmethode, grafisch weer. Dit schema herhalen wij nog eens in figuur 12-1. Met behulp van de in hoofdstuk 8 behandelde mogelijkheden voor het creëren van datatypen werkten we vervolgens de pakketspecificaties voor EENHEID_INFORMATIE, DATA_BASE en OPVRAAG_BEWERKINGEN verder uit. Rest nog het nader uitwerken van het hoofdprogramma en van de bodies van de pakketten.



Figuur 12-1 Ontwerp voor OPVRAGEN_VAN_PROJECTGEGEVENS

12.2 Programmeer De Bewerkingen



Ook in dit stadium van de uitwerking van de oplossing hebben we al een globaal inzicht in de structuur daarvan en ook waren we al in staat om die in Ada te formuleren. De mogelijkheid om al in zo'n vroeg stadium de programmeertaal zinvol te gebruiken voor het formuleren van de oplossingsmethode, is één van de grote voordelen van Ada. Bij de meeste andere programmeertalen kan pas begonnen worden met het beschrijven van de oplossing in de programmeertaal zelf, als het ontwerp volledig is uitgewerkt en als de fase van het coderen kan beginnen. Met behulp van Ada kan ook een globaal en nog in nader detail uit te werken ontwerp al in de taal zelf worden geformuleerd. Ja, het is zelfs mogelijk deze ontwerpoplossing al te compileren, uit te voeren en te testen!

Het ontwikkelen van software is een iteratief proces: hoe gedetailleerd en uitgebreid het probleem ook geanalyseerd wordt, pas tijdens de uitwerking van de oplossing komen aspecten aan het licht, die eerder over het hoofd werden gezien en die vaak opnieuw formuleren of herzien van de oplossingsmethode noodzakelijk maken. Dit is hier ook het geval: we weten wel wat het systeem in grote lijnen moet doen, maar we hebben nog onvoldoende aandacht geschonken aan de interactie van het systeem met de gebruiker. De beste aanpak is om een aantal verschillende scenario's voor opvraagsessie's op te stellen. Als ons probleem werkelijk voor de praktijk zou moeten worden uitgewerkt, dan zouden deze scenario's vervolgens met de gebruiker moeten worden geanalyseerd, totdat was vastgesteld dat aan alle gestelde eisen is voldaan. Een dergelijke *walkthrough* (letterlijk: doorlopen) van het systeem van de gebruiker samen met de programmeur, werkt vaak zeer verduidelijkend en leidt tot een systeem dat beter is aangepast bij de behoefte en wensen van de gebruiker.

Voor dit voorbeeld hebben we zelf een paar scenario's voor opvraagsessies opgesteld. Om het geheel niet al te ingewikkeld te maken zullen we afzien van de mogelijkheid dat de gebruiker onjuiste, dat wil zeggen door het programma niet te verwerken, gegevens invoert. We verwachten dus van de gebruiker dat deze altijd correct zal handelen. Dit is natuurlijk niet erg realistisch, omdat juist menselijke fouten erg vaak voorkomen. In hoofdstuk 17 zullen we laten zien, dat het opvangen van fouten in Ada met behulp van de zogenaamde 'exception handling' mogelijkheden, vrij eenvoudig is.

Om te beginnen wil een gebruiker het opvraagsysteem kunnen opstarten (vanuit de programma-omgeving) en vervolgens de handelingen kunnen uitvoeren die we eerder omschreven: een gesorteerde lijst produceren met gegevens over de programma-eenheden, informatie over een daartoe geselecteerde eenheid opvragen en statistische gegevens of kentallen produceren over de ontwikkelingsstatus van alle eenheden. Een goede en gebruikelijke methode is, de gebruiker via een 'prompt' te vragen om een commando in te tikken, en vervolgens dit commando te bevestigen. Een bepaalde sessie zou dus als volgt kunnen beginnen:

```
VOER OPDRACHT IN: produceer_lijst
PRODUCEER_LIJST COMMANDO GEACCEPTEERD
```

We schrijven de door het systeem geproduceerde tekst in hoofdletters en de door de gebruiker ingetikte tekst in kleine letters.

In figuur 12-1 staat nog eens weergegeven, dat de hoofdprogramma-eenheid, OPVRAGEN_VAN_GEGEVENS de mogelijkheden in OPVRAAG_BEWERKINGEN gebruikt om de database te benaderen. In hoofdstuk 9 formuleerden we het interface voor dit pakket aldus:

```
with PROJECT;
use PROJECT;
package OPVRAAG_BEWERKINGEN is
  type OPDRACHT is (VERZAMEL_KENTALLEN, PRODUCEER_LIJST,
                    STOP, SELECTEER_EENHEID);
  function VERZOEK return OPDRACHT;
  procedure VERZAMEL_KENTALLEN;
  procedure PRODUCEER_LIJST;
  procedure SELECTEER_EENHEID;
end OPVRAAG_BEWERKINGEN;
```

OPVRAAG_BEWERKINGEN noemt PROJECT weliswaar in zijn with-clausule, maar toch moet elke eenheid, die OPVRAAG_BEWERKINGEN gebruikt eveneens PROJECT in een with-clausule noemen, om PROJECT direct te kunnen benaderen. Dit is dus een manier om de toegang tot programma-eenheden op een lager abstractieniveau te regelen. Daarom ook zijn procedures zonder parameters gebruikt in OPVRAAG_BEWERKINGEN - het hoofdprogramma kan en mag geen directe toegang verkrijgen tot de database. OPVRAAG_BEWERKINGEN is een voorbeeld van een verzameling subprogramma's, die de benaderde objecten voor de gebruiker niet direct toegankelijk maken. In hoofdstuk 13 zullen wij hier nader op ingaan.

We kunnen nu het hoofdprogramma als volgt formuleren:

```
with OPVRAAG_BEWERKINGEN, TEXT_IO;
use OPVRAAG_BEWERKINGEN;
procedure OPVRAGEN_VAN_PROJECTGEGEVENS is
begin
  loop
    TEXT_IO.PUT("VOER OPDRACHT IN: ");
    case OPVRAAG_BEWERKINGEN.VERZOEK is
      when VERZAMEL_KENTALLEN =>
           TEXT_IO.PUT("VERZAMEL_KENTALLEN COMMANDO GEACCEPTEERD");
           TEXT_IO.NEW_LINE;
           OPVRAAG_BEWERKINGEN.VERZAMEL_KENTALLEN;
      when PRODUCEER_LIJST    =>
           TEXT_IO.PUT("PRODUCEER_LIJST COMMANDO GEACCEPTEERD");
           TEXT_IO.NEW_LINE;
           OPVRAAG_BEWERKINGEN.PRODUCEER_LIJST;
      when STOP               =>
           exit;
```

```
      when SELECTEER_EENHEID   =>
           TEXT_IO.PUT("SELECTEER_EENHEID COMMANDO GEACCEPTEERD");
           TEXT_IO.NEW_LINE;
           OPVRAAG_BEWERKINGEN.SELECTEER_EENHEID;
    end case;
  end loop;
end OPVRAAG_VAN_PROJECTGEGEVENS;
```

In bovenstaand Ada-subprogramma is exact beschreven welke acties het systeem, gezien vanuit een hoog abstractieniveau, moet uitvoeren. Voor toegang tot de database wordt het pakket OPVRAAG-_BEWERKINGEN gebruikt, voor het regelen van invoer en uitvoer wordt het voorgedefinieerde pakket TEXT_IO gebruikt (zie hoofdstuk 19). Uit het pakket TEXT_IO gebruikten we de operaties PUT en NEW_LINE om de verzoeken om het invoeren van een opdracht te produceren. Toegang tot beide pakketten (*bibliotheekeenheden* genaamd) wordt verkregen via de **with**-clausule. Deze eenheden worden daardoor zichtbaar'. *Zichtbaarheid* wil in dit verband zeggen dat de eenheid vanuit een bibliotheek wordt geïmporteerd en vervolgens benaderd kan worden. (Een eenheid die in een module wordt *geïmporteerd* kan binnen die module worden gebruikt, ook als de opdracht buiten de module is gedeclareerd.) Een eenheid heet *direct zichtbaar* als die eenheid benoemd kan worden, zonder dat een voorvoegsel hoeft te worden toegevoegd, dat aangeeft vanwaar die eenheid afkomstig is.

TEXT_IO hebben we niet via een **use**-clausule direct zichtbaar gemaakt en dat betekent dus dat de eenheden binnen TEXT_IO via de puntnotatie moeten worden benaderd, zoals is gebeurd bij TEXT_IO.PUT. Het toevoegen van de pakketnaam op deze wijze, maakt precies duidelijk waar de gebruikte functies of procedures vandaan komen en dit vergroot dus de leesbaarheid van de oplossing. Bij het pakket OPVRAAG_BEWERKINGEN hebben we wel de **use**-clausule toegepast. Dit pakket is dus wel direct zichtbaar en dit betekent dat bijvoorbeeld in de gebruikte **case**-instructie de verschillende waarden van de OPDRACHT direct kunnen worden benaderd. Het is echter niet verboden om ook in dit geval terwille van de leesbaarheid, de pakketnaam als voorvoegsel via de puntnotatie toe te voegen.

Na de contextspecificatie met behulp van de **with**- en **use**-clausule, volgt de body van het subprogramma. Het declareren van lokale objecten in het hoofdprogramma is in dit geval niet nodig: alle benodigde objecten zijn al opgenomen in OPVRAAG_BEWERKINGEN. Het hoofdprogramma bestaat uit een elementaire lus, met daarin een **case**-instructie. Aan het begin van de **case** wordt het subprogramma OPDRACHT uit OPVRAAG_BEWERKINGEN aangeroepen om de opdracht van de gebruiker in te voeren. Op grond van de evaluatie in de **case**-instructie wordt vervolgens de bijbehorende actie uitgevoerd. Is de opdracht STOP dan wordt de lus verlaten en wordt de verwerking van het hoofdprogramma beëindigd.

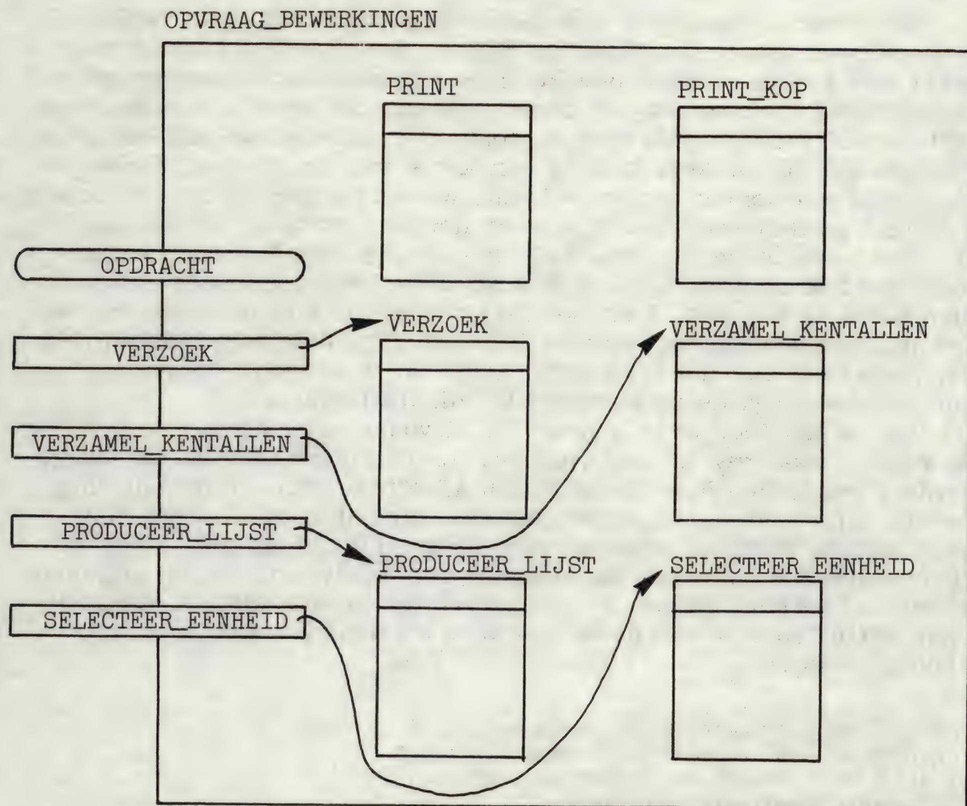
De communicatie van OPVRAAG_BEWERKINGEN is nu vastgelegd, maar nu moeten de operaties zelf nog worden uitgevoerd. Volgens de regels van Ada moeten de operaties, die in een zichtbaar gedeelte van een pakket worden gedefinieerd, in de bijbehorende pakketbody worden uitgewerkt. Het raamwerk voor de body van OPVRAAG-_BEWERKINGEN ziet er nu zo uit:

```
with TEXT_IO;
use TEXT_IO;
package body OPVRAAG_BEWERKINGEN is
--
   -- lokale pakketeenheden (utilities) worden hier geplaatst
   --
   function VERZOEK return OPDRACHT is
      . . .
   end VERZOEK;
   procedure VERZAMEL_KENTALLEN is
      . . .
   end VERZAMEL_KENTALLEN;
   procedure PRODUCEER_LIJST is
      . . .
   end PRODUCEER_LIJST;
   procedure SELECTEER_EENHEID is
      . . .
   end SELECTEER_EENHEID;
end OPVRAAG_BEWERKINGEN;
```

Het eerste deel van bovenstaande pakketbody wordt gereserveerd voor een aantal algemeen bruikbare gereedschappen of utilities. Bij de verdere uitwerking van de vier belangrijkste routines van het pakket zal blijken, dat er een aantal routines voor gemeenschappelijk gebruik op een lager niveau kunnen worden ontwikkeld. In figuur 12-2 wordt de structuur van ons OPVRAAG_BEWERKINGEN pakket nog eens nader geillustreerd.

Volgens de regels voor pakketten in Ada is alles dat zichtbaar is in de specificatie van een pakket, ook zichtbaar in de bijbehorende pakketbody. Om een voorbeeld te geven: PROJECT is zichtbaar in de programmabody, omdat we PROJECT noemden in de contextspecificatie (**with** en **use**) van het pakket. Zo noemden we ook TEXT_IO, waardoor het mogelijk wordt de routines uit TEXT_IO voor communicatie met de buitenwereld te gebruiken. In de specificatie van OPVRAAG_BEWERKINGEN (zie hoofdstuk 9), noemden we TEXT_IO niet; immers daar was gebruik van faciliteiten uit TEXT_IO niet noodzakelijk.

We willen uitdrukkelijk wijzen op de volgende stelregel: importeer *alleen* die faciliteiten in een eenheid, die ook daadwerkelijk door die eenheid worden gebruikt. Meer invoeren dan nodig is, maakt de programmatuur hopeloos complex. Grootheden die binnen een pakketbody worden gedeclareerd, zijn van buiten deze body niet toegankelijk; zij zijn *verborgen* ('hidden').



Figuur 12-2 Ontwerp voor OPVRAAG_BEWERKINGEN

We gaan nu verder met de uitwerking van de pakketbody. Bij de formulering van de functie VERZOEK gaan we er weer vanuit dat er alleen correcte opdrachten door de gebruiker worden gegeven: dat wil zeggen alleen de opdrachten VERZAMEL_KENTALLEN, PRODUCEER_LIJST, STOP, of SELECTEER_EENHEID. De VERZOEK-functie kan er nu als volgt uitzien:

```
function VERZOEK return OPDRACHT is
  GEBRUIKERSOPVRAAG : OPDRACHT;
  package OPDRACHT_IO is new
          TEXT_IO;ENUMERATION_IO(OPDRACHT);
begin
  OPDRACHT_IO.GET(GEBRUIKERSOPVRAAG);
  TEXT_IO.NEW_LINE;
  return GEBRUIKERSOPVRAAG;
end VERZOEK;
```

Het was hier nodig om een lokaal object GEBRUIKERSOPVRAAG te declareren binnen de functie. Binnen functies in Ada moet vaak eerst een waarde worden berekend en vervolgens geretourneerd via **return**; dan moet er ook een object zijn dat die waarde kan bevatten. In het declaratiegedeelte van ons pakket creëerden we ook een eigen OPDRACHT_IO pakket, in feite een kopie van de voorgedefinieerde Input/Output voor enumeratietypen. We verkrijgen op deze wijze een generiek pakket voor het type waar het hier om gaat: OPDRACHT. Dit kopiëren, in het Engels *instantanation* genoemd, wordt verder behandeld in hoofdstuk 14. Ada's Input/outputprocedures komen in hoofdstuk 19 aan bod. Laat het hier voldoende zijn te weten dat we met het creëren van het pakket OPDRACHT_IO de mogelijkheid hebben, waarden van het type OPDRACHT in te voeren. Dit gebeurt dan met name via de procedure GET uit dit pakket.

Nu we het toch over invoer- en uitvoerroutines hebben, bekijken we meteen maar een andere routine, waarbij communicatie met de gebruiker nodig is. Voor PRODUCEER_LIJST werd in de formulering van de informele strategie aangegeven dat het mogelijk moet zijn gegevens te sorteren op naam van de eenheid, of op grond van de specificatie-identificatie. We zullen veronderstellen, dat de gegevens worden afgedrukt op een standaardprinter in alfabetische volgorde. Voor het uitvoeren van de opdracht is nu bijvoorbeeld de volgende dialoog nodig:

```
VOER OPVRAAGOPDRACHT IN: produceer_lijst
PRODUCEER_LIJST OPDRACHT IS GEACCEPTEERD
OP WELK VELD WILT U SORTEREN? eenheid
SORTERING WORDT UITGEVOERD . . .

NAAM EENHEID     NAAM PROGRAMMEUR  STATUS     VERSIE  SPECIFICATIES
MEET_GEGEVENS    BABBAGE, CH.      ONTWERP      1        CPCI 7
STUUR_ROUTINES   LOVELACE, A.      CODERING     6        CPCI 4
. . .            . . .             . . .      . . .      . . .
```

Zou de gebruiker 'specificatie' hebben ingetikt, dan moeten de gegevens gesorteerd worden op specificatie-identificatie (SPECIFICATIE-_DOCUMENT).

We hebben in PRODUCEER_LIJST nog enkele objecten nodig voor de verdere uitwerking van de oplossing. Zo is er een object nodig dat de sorteercriteria aangeeft: SORTEERCRITERIA. We declareren dit object als een enumeratietype en sommen dus alle mogelijke waarden expliciet op (dit maakt het ook makkelijk later nieuwe criteria toe te voegen). Het invoeren van een sorteercriterium zal mogelijk worden gemaakt via de creatie van een generieke kopie van het pakket TEXT_IO. De gegevens zelf zijn bereikbaar via het pakket DATA_BASE en heten DATA_BASE.DATA.

Dan is er nog een bewerking nodig voor het sorteren van de gegevens: SORT. Omdat de database op den duur misschien erg groot kan worden, geven we niet de hele database mee als parameter voor de SORT-operatie; het resultaat van SORT wordt een lijst van recordnummers, die verwijzen naar de records van de database en die in de volgorde van de gewenste sortering staan.

De functie PRINT zal de gesorteerde gegevens afdrukken. Omdat in de informele strategie vermeld staat, dat er ook andere opvraagfuncties kunnen zijn die het afdrukken van de gegevens over een bepaalde unit vereisen, zullen we PRINT declareren als een lokaal subprogramma binnen het pakket. Dit is dus een utility subroutine. De PRINT routine kan er nu zo uitzien:

```
procedure PRINT(DATA_RECORD: in EENHEID_INFORMATIE.EENHEID RECORD) is
  package STATUS_IO is new
            TEXT_IO.ENUMERATION_IO(EENHEID_INFORMATIE.STATUS_TYPE);
  package VERSIE_IO is new
            TEXT_IO.INTEGER_IO(EENHEID_INFORMATIE.VERSIE_TYPE);
  use STATUS_IO,VERSIE_IO;
begin
  SET_COL(TO => 1);
  PUT(DATA_RECORD.EENHEID_NAAM,WIDTH => 20);
  SET_COL(TO => 21);
  PUT(DATA_RECORD.PROGRAMMEUR,WIDTH => 20);
  SET_COL(TO => 42);
  PUT(DATA_RECORD.STATUS,WIDTH => 11);
  SET_COL(TO => 54);
  PUT(DATA_RECORD.VERSION,WIDTH => 5);
  SET_COL(TO => 62);
  PUT(DATA_RECORD.SPECIFICATIE.DOCUMENT,WIDTH => 80);
  NEW_LINE;
end PRINT;
```

In deze PRINT-routine worden een aantal procedures uit het voorgedefinieerde pakket TEXT_IO gebruikt. Voor de lay-out op de regel wordt bijvoorbeeld de procedure SET_COL gebruikt, om de plaats aan te geven waar het volgende gegeven moet worden afgedrukt.

Boven de output plaatsen we een titel met behulp van een routine PRINT_KOP:

```
procedure PRINT_KOP is
begin
  SET_COL(TO => 1);
  PUT("NAAM EENHEID");
  SET_COL(TO => 21);
  PUT("NAAM PROGRAMMEUR");
  SET_COL(TO => 42);
  PUT("STATUS");
  SET_COL(TO => 54);
  PUT("VERSIE");
  SET_COL(TO => 62);
  PUT("SPECIFICATIES");
  NEW_LINE;
end PRINT_KOP;
```

Nu deze objecten en operaties op het lagere abstractieniveau zijn beschreven, kunnen we het PRODUCEER_LIJST subprogramma uitwerken:

```
procedure PRODUCEER_LIJST is
  type SORTEERSLEUTEL is (SPECIFICATIE,EENHEID);
  SORTEERCRITERIA : SORTEERSLEUTEL;
  package CRITERIA_IO is new TEXT_IO.ENUMERATION_IO(SORTEERSLEUTEL);
  use CRITERIA_IO;
  type SORTEERLIJST is array(DATA_BASE.RECORD_INDEX)
                      of DATA_BASE.RECORD_INDEX;
  SORTEERTABEL : SORTEERLIJST;
procedure SORTEER(RECORDS : in   DATA_BASE.PROJECT_RECORDS;
                  OMVANG  : in   DATA_BASE.RECORD_INDEX;
                  CRITERIA: in   SORTEERSLEUTEL;
                  LIJST   : out  SORTEERLIJST) is separate;
begin
  PUT("OP WELK VELD WILT U SORTEREN? ");
  GET(SORTEERCRITERIA);
  NEW_LINE;
  SORTEER(RECORDS => DATA_BASE.DATA,
          OMVANG  => DATA_BASE.ACTIEVE_RECORDS,
          CRITERIA => SORTEERCRITERIA,
          LIJST    => SORTEERTABEL);
  PUT("SORTERING WORDT UITGEVOERD ... ");
  NEW_LINE;
  PRINT_KOP;
  for INDEX in 1 .. DATA_BASE.ACTIEVE_RECORDS
    loop
      PRINT(DATA_BASE.DATA(SORTEERTABEL(INDEX)));
    end loop;
end PRODUCEER_LIJST;
```

De aanroep van PRINT ziet er op het eerste gezicht wat ingewikkeld uit; we gebruikten hier een combinatie van de puntnotatie en indexering. We PRINTen een DATA_BASE.DATA record, namelijk dat record, waarnaar door SORTEERTABEL(INDEX) verwezen wordt.

Verder valt op te merken, dat we in het declaratiegedeelte een procedure SORTEER introduceerden, waarvan de body niet wordt uitgewerkt. Tijdens het top-down ontwerp van PRODUCEER_LIJST wordt SORTEER gezien als een elementaire operatie: hoe die precies wordt uitgevoerd is hier niet interessant. Trouwens ook de leesbaarheid van het geheel zou er niet op vooruitgaan als we de SORTEER-procedure hier zouden uitschrijven: het aantal regels zou minstens verdubbelen! Door SORTEER als **separate** te declareren is de interface met de eenheid zichtbaar, maar de uitwerking van de algoritme wordt afzonderlijk gecompileerd. Dit heeft nog een voordeel: als blijkt dat er een efficiëntere oplossing voor de SORTEER-algoritme mogelijk is, dan kan de body van SORTEER worden vervangen, zonder dat er in PRODUCEER_LIJST of welke andere programma-eenheid dan ook, iets hoeft te worden veranderd.

Een mogelijke oplossing is een index datastructuur op te bouwen, die de alfabetische volgorde van DATA_BASE.DATA aangeeft. Dit behoeft dan maar één keer te gebeuren en de SORTEER-procedure bestaat dan uit niets anders dan uit het opvragen van deze index. Hoewel deze oplossing te verkiezen is, laten we hier een wat minder efficiënte oplossing zien, die echter het gebruik van besturingsstructuren in Ada nog wat nader illustreert. We gebruiken de

eenvoudige insertie sorteermethode (telkens tussenvoegen van het volgende element op de juiste plaats) om de DATA_BASE.DATA te sorteren en met als resultaat een LIJST van RECORD_INDEXen, die in dit geval oplopend is van 1 .. OMVANG, omdat de records daadwerkelijk fysiek gesorteerd worden:

```
separate (PRODUCEER_LIJST)
procedure SORTEER(RECORDS : in  DATA_BASE.PROJECT_RECORDS;
                  OMVANG  : in  DATA_BASE.RECORD_INDEX;
                  CRITERIA: in  SORTEERSLEUTEL;
                  LIJST   : out SORTEERLIJST)
  TEMP_RECORD : EENHEID_INFORMATIE.EENHEID_RECORD;
function NIET_OP_VOLGORDE(EERSTE   : in EENHEID_INFORMATIE.EENHEID_RECORD;
                          TWEEDE   : in EENHEID_INFORMATIE.EENHEID_RECORD;
                          CRITERIA: in SORTEERSLEUTEL) return BOOLEAN is
begin
  case CRITERIA is
    when EENHEID        => if EERSTE.EENHEID_NAAM >TWEEDE.EENHEID_NAAM then
                             return TRUE;
                           else
                             return FALSE;
                           end if;
    when SPECIFICATIE => if EERSTE.SPECIFICATIE_DOCUMENT >
                             TWEEDE.SPECIFICATIE_DOCUMENT then
                             return TRUE;
                           else
                             return FALSE;
                           end if;
  end case;
end NIET_OP_VOLGORDE;
begin
  for EERSTE_INDEX in 1 .. OMVANG - 1
    loop
      for TWEEDE_INDEX in  EERSTE_INDEX + 1 .. OMVANG
        loop
          if NIET_OP_VOLGORDE(RECORDS(EERSTE_INDEX),
                              RECORDS(TWEEDE_INDEX),CRITERIA) then
            TEMP_RECORD            := RECORDS(EERSTE_INDEX);
            RECORDS(EERSTE_INDEX) := RECORDS(TWEEDE_INDEX);
            RECORDS(TWEEDE_INDEX) := TEMP_RECORD;
          end if;
        end loop;
    end loop;
for INDEX in 1 .. OMVANG
  loop
    LIJST(INDEX) := INDEX;
  end loop;
end SORTEER;
```

In de allereerste regel van deze programma-eenheid wordt aangegeven bij welke eenheid deze afzonderlijk gecompileerde eenheid behoort (namelijk bij PRODUCEER_LIJST). SORTEER is in feite een subeenheid en in hoofdstuk 20 zullen we nader uiteenzetten hoe subelementen bij top-down ontwerp van pas kunnen komen. In de body van dit subprogramma wordt een lokale functie gedeclareerd: NIET_OP_VOLGORDE, die bepaalt of twee records op grond van

CRITERIA verwisseld moeten worden. De invoeg-sorteermethode zelf wordt uitgevoerd met behulp van een paar geneste lussen. Twee niet op volgorde staande records worden onderling verwisseld. Dit proces wordt voortgezet, totdat de gehele rij is gesorteerd en tenslotte krijgen de elementen LIJST de waarde van hun INDEX als waarde toegekend (dus LIJST(1) = 1, LIJST(2) = 2, enzovoort).

Nogmaals: dit fysieke sorteren van de records is weinig efficiënt en het alleen opbouwen van een gesorteerde index was veel handiger geweest. Omdat echter de interface met SORTEER louter plaatsvindt via het object LIJST, kan deze efficiëntere methode gemakkelijk worden ingevoerd, zonder dat de rest van de oplossing enige wijziging hoeft te ondergaan. Dit is weer een illustratie van het nut van het pakketmechanisme in Ada: systemen blijven onderhoudbaar en de effecten van wijzigingen blijven beperkt.

We gaan nu het subprogramma SELECTEER_EENHEID uitwerken, dat alle gegevens over een bepaalde eenheid afdrukt. Als de opdracht SELECTEER_EENHEID wordt gegeven, dan zou bijvoorbeeld de volgende dialoog kunnen plaatsvinden:

```
GEEF OPVRAAG COMMANDO: selecteer_eenheid
SELECTEER_EENHEID COMMANDO GEACCEPTEERD
OVER WELKE EENHEID WILT U GEGEVENS? stuur_routines
SELECTIE VAN GEGEVENS VIA EENHEIDNAAM WORDT UITGEVOERD ...

NAAM EENHEID      NAAM PROGRAMMEUR  STATUS     VERSIE  SPECIFICATIES
STUUR_ROUTINES    LOVELACE, A.      CODERING     6        CPCI 4
```

Het SELECTEER subprogramma wordt dan bijvoorbeeld:

```
procedure SELECTEER_EENHEID is
  EENHEID_NAAM : EENHEID_INFORMATIE.NAAM_TYPE;
begin
  PUT("OVER WELKE EENHEID WILT U GEGEVENS? ");
  GET(EENHEID_NAAM);
  NEW_LINE;
  PUT("SELECTIE VAN GEGEVENS VIA EENHEIDNAAM WORDT UITGEVOERD ... ");
  NEW_LINE;
  PRINT_KOP
  for INDEX in 1 .. DATA_BASE_ACTIEVE_RECORDS
    loop
      if DATA_BASE.DATA(INDEX).EENHEID_NAAM = EENHEID_NAAM then
        PRINT(DATA_BASE.DATA(INDEX));
        exit;
      end if;
    end loop;
end SELECTEER_EENHEID;
```

U ziet hoe de exit-instructie wordt gebruikt om de lus te verlaten, zodar de EENHEID_NAAM wordt gevonden.

Tenslotte moet er nog een bewerking in OPVRAAG_BEWERKINGEN worden geschreven, namelijk het subprogramma VERZAMEL_KENTALLEN. We gaan uit van de volgende dialoog met het programma:

```
GEEF OPVRAAG COMMANDO: verzamel_kentallen
VERZAMEL_KENTALLEN COMMANDO GEACCEPTEERD
KENTALLEN WORDEN VERZAMELD ...
STATUS          ABSOLUTE FREQUENTIE   RELATIEVE FREQUENTIE
ONTWERP                  40                   44.4
CODERING                 26                   28.9
TEST                     17                   18.9
OPERATIONEEL              7                    7.8
```

De gekozen oplossing doorloopt eenvoudig alle DATA_BASE.DATA en telt hoe vaak eenheden in een bepaalde status voorkomen. Vervolgens wordt de relatieve frequentie per status berekend en worden de resultaten afgedrukt:

```
procedure VERZAMEL_KENTALLEN is
  package STATUS_IO is new
          TEXT_IO.ENUMERATION_IO(EENHEID_INFORMATIE.STATUS_TYPE);
  use STATUS_IO;
  type ABSOLUUT is range 0 .. DATA_BASE.MAXIMUM_RECORDS;
  package ABSOLUUT_IO is new TEXT_IO.INTEGER_IO(ABSOLUUT);
  use ABSOLUUT_IO;
  type RELATIEF is delta 0.1 range 0.0 .. 100.0;
  package RELATIEF_IO is new TEXT_IO.FIXED_IO(RELATIEF);
  use RELATIEF_IO;
  type STATUS RECORD is record
                          TOTAAL     : ABSOLUUT;
                          PERCENTAGE : RELATIEF;
                        end record;
  type STATUS_DATA is array (EENHEID_INFORMATIE.STATUS_TYPE) of
                          STATUS_RECORD;
  FREQUENTIE : STATUS_DATA := (ONTWERP .. OPERATIONEEL =>
                              (TOTAAL => 0, PERCENTAGE => 0.0));
begin
  PUT("KENTALLEN WORDEN VERZAMELD ... ");
  NEW_LINE;
  PUT("STATUS   ABSOLUTE FREQUENTIE   RELATIEVE FREQUENTIE");
  NEW_LINE;
  for INDEX in 1 .. DATA_BASE.ACTIEVE_RECORDS
    loop
      FREQUENTIE(DATA_BASE.DATA(INDEX).STATUS).TOTAAL :=
             FREQUENTIE(DATA_BASE.DATA(INDEX).STATUS).TOTAAL + 1;
    end loop;
  for INDEX in EENHEID_INFORMATIE.STATUS_TYPE
    loop
      FREQUENTIE(INDEX).PERCENTAGE :=
             RELATIEF(FLOAT(FREQUENTIE(INDEX).TOTAAL))/
             DATA_BASE.ACTIEVE_RECORDS;
      PUT(INDEX,WIDTH => 11);
      PUT("      ");
      PUT(FREQUENTIE(INDEX).TOTAAL);
      PUT("                ");
      PUT(FREQUENTIE(INDEX).PERCENTAGE);
      NEW_LINE;
    end loop
end VERZAMEL_KENTALLEN;
```

Hiermee zijn we klaar met de uitwerking van het pakket OPVRAAG_BEWERKINGEN en we bekijken nu nog even het DATA-_BASE pakket. Telkens wanneer dit pakket zichtbaar wordt door het gebruik van een **with** clausule, dan creëren we in feite een eigen kopie van het pakket. Bedenk echter dat, afgezien van access-objecten, Ada geen beginwaarden aan objecten toekent. We zullen dus nog een stukje programma moeten schrijven dat DATA_BASE.DATA met waarden vult. Een pakket dat alleen maar typen en objecten exporteert hoeft eigenlijk geen pakketbody te hebben; in dit geval voegen wij echter wel een body toe met een aantal instructies, die bij de uitwerking van de body worden uitgevoerd.

Gebruikmakend van een paar voorgedefinieerde file I/O subprogramma's (zie hoofdstuk 19), kan het pakket nu als volgt worden voltooid:

```
with SEQUENTIAL_IO;
package body DATA_BASE is
  package BASE_IO is new SEQUENTIAL_IO(EENHEID_RECORD);
  use BASE_IO;
  DATA_FILE : BASE_IO.IN_FILE;
begin
  ACTIEVE_RECORDS := 0;
  OPEN(DATA_FILE,
        MODE => IN_FILE,
        NAME => "PROJECT_A/EENHEDEN");
  while not END_OF_FILE(DATA_FILE)
    loop
      ACTIEVE_RECORDS := ACTIEVE_RECORDS + 1;
      READ(DATA_FILE,ITEM => DATA(INDEX));
    end loop;
  CLOSE(DATA_FILE);
end DATA_BASE;
```

Bij verwerking van deze eenheid wordt eerst de file "PROJECT_A/ EENHEDEN" geopend en vervolgens worden de records één voor één ingelezen. Als het einde van de file is bereikt, dan wordt deze weer gesloten, om het programma netjes te laten aflopen.

We zijn nu klaar met ons database opvraagsysteem. Alle behandelde programma-eenheden zijn afzonderlijk compileerbaar, maar om nog eens een algeheel overzicht te krijgen is in Appendix F een volledige listing opgenomen. Hopelijk begint u het nut van Ada's pakketmechanisme in te zien en hebt u op grond van dit voorbeeld op zijn minst een intuïtief inzicht in het gebruik ervan. We hebben nog een aantal onderdelen van het pakketmechanisme niet besproken en daarom zal dit onderwerp in het volgende hoofdstuk nog eens op een wat systematischer wijze worden behandeld.

Oefeningen

1. Wat zou er gebeuren als een gebruiker bij een aanroep van VERZOEK een niet toegelaten commando zou intikken?

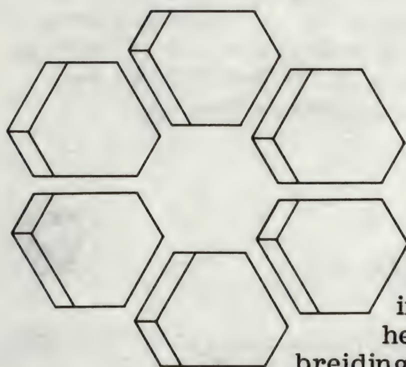
2. Herschrijf het programma PRODUCEER_LIJST zodanig, dat aan PRINT een parameter wordt gestuurd, die SORTEERCRITERIA aangeeft. Pas vervolgens PRINT aan, zodat in de eerste kolom de sorteersleutel wordt afgedrukt.

*3. Schrijf een nieuwe SORTEER-routine, die DATA_BASE.DATA niet fysiek van plaats doet veranderen.

*4. Pas het subprogramma SELECTEER_EENHEID zo aan, dat een gebruiker een eenheid kan selecteren op grond van ofwel PROGRAMMEUR, ofwel STATUS.

Pakket 5

DE PAKKET AANPAK



De mogelijkheid om nieuwe grootheden op een hoger niveau te kunnen definiëren in termen van eerder gecreëerde grootheden zou duidelijk een krachtige uitbreiding betekenen van elke programmeertaal. Op die manier zou het bouwsteenidee een onderdeel van de taal worden. In plaats van gebruik te moeten maken van een vast repertoire van instructies, waaruit ieder programma zou moeten worden opgebouwd, zou de programmeur nu zijn eigen modules kunnen construeren, elk met een eigen naam en overal binnen het programma bruikbaar, alsof het ging om in de taal ingebouwde mogelijkheden.

D.R. Hofstadter
Gödel, Escher, Bach:
An Eternal Golden Braid [1]

13 PAKKETTEN

Veronderstel dat u thuis een lekkage heeft en een reparatie moet uitvoeren. Om te beginnen zou u dat kunnen proberen met het materiaal dat u toevallig in huis hebt - om drie uur in de morgen is bijna alles geschikt om een lek te stoppen! Verstandiger is uiteindelijk als u toch 's morgens naar de gereedschappenwinkel stapt en de spullen koopt die nodig zijn voor een vakkundige en afdoende reparatie. Thuis legt u dan het geschikte gereedschap bij elkaar en gaat aan het werk.

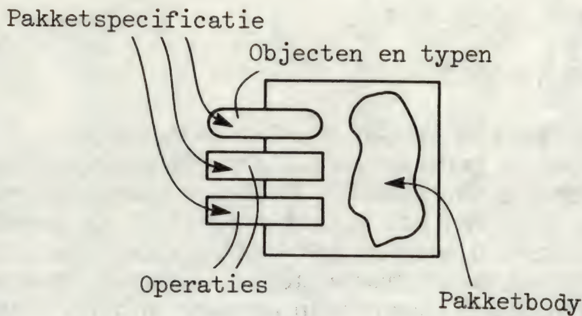
Nu stopt Ada wel geen lekkende kranen, maar toch is er een overeenkomst. In Ada kan de programmeur softwaregereedschappen verzamelen in een *pakket*. In de eerste twee ontwerpproblemen gebruikten we al pakketten bij de formulering van de oplossing; een intuïtief begrip voor deze taalmogelijkheid zult u dus wel al hebben. We gaan nu pakketten in Ada ('packages') nader bekijken en we gaan in op hun structuur en hun toepassingsmogelijkheden.

13.1 Pakketten In Ada: Vorm



Een pakket bestaat uit een collectie logisch samenhangende eenheden of rekengereedschappen. Een pakket *kapselt* die eenheden *in* (bouwt er als het ware een muur omheen). In hoofdstuk 6 tekenden we het symbool voor het Ada-pakket, in figuur 13-1 herhalen we dit nog eens. We zien dat een pakket uit twee gedeelten bestaat: de specificatie en de body ('romp'). De specificatie beschrijft de zichtbare onderdelen van het pakket; deze onderdelen worden door het pakket 'geëxporteerd'. Voor geëxporteerde objecten en typen gebruikten we de rechthoek met afgeronde hoeken; voor operaties die met rechte hoeken.

De pakketspecificatie is in zekere zin het contract tussen de maker van het pakket en de gebruiker daarvan. Dit interface of communicatiepatroon geeft aan welke onderdelen van het pakket gebruikt mogen worden en op welke wijze dat moet gebeuren. De gebruiker hoeft daarbij niet te weten hoe de bewerkingen precies zijn geprogrammeerd. De pakketgebruiker kan trouwens alleen



Figuur 13-1 Symbolische weergave van het Ada-pakket

naar de van buitenaf zichtbare onderdelen verwijzen. In een auto bijvoorbeeld, bestaan de communicatiemiddelen met de gebruiker uit het stuurwiel, de rem en het gaspedaal; dit zijn de zichtbare hulpmiddelen. De bestuurder hoeft niet te weten hoe ze werken - dat is de technische uitwerking. Op overeenkomstige wijze kan in Ada de uitwerking van mechanismen verborgen blijven binnen de body van een pakket.

De pakketstructuur bevordert direct ontwerpprincipes als modulariteit, abstractie, lokaal houden en beperkt toegankelijk maken van gegevens. Natuurlijk zullen goede programmeurs deze principes net zo goed toepassen als zij programmeren in een andere programmeertaal, zoals FORTRAN of zelfs assembleertaal. Ada onderscheidt zich van de meeste andere programmeertalen, omdat Ada die principes bijna bindend oplegt en in ieder geval sterk bevordert. De taalregels zorgen ervoor, dat een pakketgebruiker precies dat kan doen wat de pakketspecificatie toelaat en niets meer. Probeert de gebruiker dat toch, dan wordt dat tijdens de compilatie afgestraft door foutmeldingen.

Specificatie en body kunnen afzonderlijk worden gecompileerd. Dit maakt het mogelijk de specificatie in een vroeg stadium van het ontwerpproces in te voeren; de body kan dan later worden uitgewerkt. Trouwens, de fysieke scheiding van specificatie en body is nog eens een letterlijke vorm van 'information hiding'. Maar het allerbelangrijkste is wel, dat de Ada pakketten de programmeur helpen bij het onder controle houden van ingewikkelde oplossingen, omdat het pakket een mechanisme biedt met behulp waarvan onderling samenhangende grootheden op logische wijze gegroepeerd kunnen worden tot bouwstenen.

Pakketspecificaties

Een pakketspecificatie ziet er zo uit:

```
package NOEM_MAAR_WAT is
    . . .
end NOEM_MAAR_WAT
```

De pakketspecificatie kan verder worden onderverdeeld in twee gedeelten: het *zichtbare* gedeelte en het *private* gedeelte. Het zichtbare deel declareert de hulpmiddelen die van buiten het pakket benaderd kunnen worden; het pakket *exporteert* die grootheden. Een pakket kan allerleid soorten grootheden exporteren: objecten, typen, subtypen, subprogramma's, taken, getallen, excepties, constanten, herbenoemde grootheden en zelfs andere pakketten. Een goede gewoonte is om pakketspecificaties niet te lang te maken en om alleen één enkel logisch samenhangend gedeelte te exporteren. Op de manier waarop dit het beste kan gebeurèn komen we later in dit hoofdstuk nog terug.

Het private gedeelte kan alleen aan het einde van een pakketspecificatie staan en begint met het gereserveerde woord private. De tekst van het private deel is wel voor de gebruiker toegankelijk en bestaat net zoals het zichtbare deel uit een aantal specificaties. Het private deel kan echter niet van buiten het pakket worden benaderd. Het mechanisme voor afzonderlijke compilatie maakt gebruik van deze private-specificatie en in een afzonderlijke paragraaf komen we hier nog op terug.

Een programma-eenheid kan de hulpmiddelen uit elk zichtbaar pakket gebruiken. Bekijk bijvoorbeeld nog eens het pakket COMPLEX, dat we in hoofdstuk 4 invoerden:

```
package COMPLEX is
  type GETAL is record
                  REEEL_DEEL      : FLOAT;
                  IMAGINAIR_DEEL  : FLOAT;
                end record;
  function "+"(A,B : in GETAL) return GETAL;
  function "-"(A,B : in GETAL) return GETAL;
  function "*"(A,B : in GETAL) return GETAL;
end COMPLEX;
```

Het pakket complex is zichtbaar voor een programma-eenheid *P* als COMPLEX gedeclareerd wordt binnen *P*, of binnen een blok dat *P* omvat, als het pakket tenminste niet door een nieuwe declaratie verborgen wordt. (In hoofdstuk 20 worden de regels voor de reikwijdte of 'scope' van grootheden in detail behandeld.) Een voorbeeld:

```
procedure HOOFDPROGRAMMA is
  procedure EERSTE is ... end EERSTE;
                              -- begin declaratiegedeelte
  package COMPLEX is ... end COMPLEX;
                              -- specificatie van het pakket
  package body COMPLEX is ... end COMPLEX;
                              -- body van het pakket
  procedure TWEEDE is ...     -- nog een declaratie
    procedure DERDE is ... end DERDE;
                              -- een geneste procedure
  end TWEEDE;
begin
  --
  -- rij instructies van hoofdprogramma
end HOOFDPROGRAMMA;
```

COMPLEX is in het HOOFDPROGRAMMA zichtbaar vanaf het punt waar COMPLEX voor het eerst wordt genoemd. Op grond van de reikwijdte en zichtbaarheidsregels van Ada kan EERSTE, COMPLEX niet zien; TWEEDE, DERDE en het HOOFDPROGRAMMA kunnen echter wel van de mogelijkheden van COMPLEX gebruik maken.

Een andere en eigenlijk betere manier om zichtbaarheid te bewerkstelligen is de with-clausule. Het pakket COMPLEX kan afzonderlijk worden gecompileerd en kan dan als volgt door andere programma-eenheden worden gebruikt:

```
with COMPLEX;
procedure HOOFDPROGRAMMA is ...
```

HOOFDPROGRAMMA *importeert* het pakket COMPLEX. COMPLEX is nu zichtbaar in het hele HOOFDPROGRAMMA en zowel het type GETAL als de drie functies kunnen worden gebruikt. Deze benadering bevordert een modulaire opbouw en houdt effecten van wijzigingen tijdens de onderhoudfase lokaal. Een voordeel van wat subtielere aard is nog, dat de overdraagbaarheid van software-eenheden op deze manier wordt bevorderd. Over niet al te lange tijd kunt u naar de softwaregereedschappenhandel op de hoek gaan en de pakketten kopen, die u voor uw toepassing nodig hebt. De Ada pakketaanpak maakt zonder twijfel de ontwikkeling van een industrie voor overdraagbare softwarecomponenten mogelijk.

Is een pakket eenmaal zichtbaar, dan kunnen onderdelen uit de specificatie worden benaderd via de puntnotatie:

```
with COMPLEX;
procedure EEN_PROGRAMMA is
  GETAL_1,GETAL_2 : COMPLEX.GETAL;
begin ... end EEN_PROGRAMMA;
```

Door de namen die we kozen is de declaratie van GETAL_1 en GETAL_2 volkomen leesbaar en begrijpelijk. Op dezelfde manier kunnen we gebruiken:

```
GETAL_1.IMAGINAIR_DEEL := 37.961;
GETAL_1 := COMPLEX."+"(GETAL_1,GETAL_2);
```

Het laatste voorbeeld ziet er wat vreemd uit: de optellingsoperator is niet direct toegankelijk, maar moet via de puntnotatie worden aangeroepen. Ook zou de infixnotatie wat gebruikelijker zijn dan de hier gebruikte prefixnotatie, terwijl we directe toegankelijkheid zouden kunnen bewerkstelligen door de use-clausule te gebruiken. In dit laatste geval hoeft de pakketnaam COMPLEX niet meer als voorvoegsel te worden gebruikt (als de gebruikte component tenminste niet voor meer dan één uitleg vatbaar is). We krijgen dan:

```
with COMPLEX;
procedure NOG_EEN_PROGRAMMA is
  use COMPLEX;
  GETAL_3,GETAL_4 : GETAL;
begin
  GETAL_3 := GETAL_3 + GETAL_4;
end NOG_EEN_PROGRAMMA;
```

De optellingsoperator heeft nu betrekking op de door het pakket COMPLEX geëxporteerde operator en niet op de voorgedefinieerde operator voor numerieke typen. De optellingsoperator is verder direct zichtbaar, dus kan de infixnotatie worden gebruikt en kunnen de aanhalingstekens worden weggelaten. De optellingsoperator is nu *overladen*, en zolang de compiler kan uitvinden welke operator bedoeld wordt (bijvoorbeeld via de actuele parameters van de aanroep), kan zo'n overladen operator naar believen worden gebruikt. Nog een andere bijzonderheid: het is in Ada toegelaten een aantal objecten tegelijkertijd te declareren (in dit geval GETAL_3 en GETAL_4) in een zogenaamde *namenlijst* ('identifier list'). Toch wordt als regel aanbevolen maar één objectnaam per declaratie te gebruiken ter bevordering van de leesbaarheid.

De use-clausule is soms handig, omdat de namen er korter door worden, maar gebruik ervan kan ook onduidelijkheid tot gevolg hebben en zelfs interne tegenspraak tussen namen: twee of meer grootheden met dezelfde naam op hetzelfde niveau. Na gebruik van de use-clausule is het trouwens niet verboden de pakketnaam als voorvoegsel te gebruiken: GETAL_3 en GETAL_4 kunnen nog steeds als COMPLEX.GETAL worden gedeclareerd en dit verbetert de leesbaarheid. Algemene regel blijft: vermijd het onnodig gebruik van de use-clausule, teneinde de totale hoeveelheid direct zichtbare namen steeds zo klein mogelijk te houden.

Pakketbodies

De romp of body van een pakket heeft de volgende vorm:

```
package body NOEM_MAAR_EEN_NAAM is
  . . .
end NOEM_MAAR_EEN_NAAM;
```

Vanzelfsprekend moet de naam van de body volledig overeenkomen met de naam van de bijbehorende specificatie. Een body kan alleen dan desgewenst worden weggelaten als de specificatie alleen maar typen en objecten bevat en dus geen procedures. De inhoud van een pakketbody is van buitenaf niet zichtbaar en er is dus sprake van 'information hiding'.

Een pakketbody ziet er vrijwel net zo uit als een subprogramma. De body bestaat uit een declaratiedeel, al of niet gevolgd door een blok dat bestaat uit een rij instructies en eventueel een beschrijving van wat er moet gebeuren in het geval van fouten (een 'exception handler'). Als er subprogramma's, taken of pakketspecificaties in een pakketspecificatie worden geïntroduceerd, dan moeten de bodies van die grootheden in het specificatiegedeelte van de pakketbody worden uitgewerkt. (De enige uitzondering is als sprake is van subunits. Zie daarvoor hoofdstuk 20.) Net zoals in subprogrammabodies, mogen er ook weer lokale declaraties en lokale programma-eenheden worden vermeld in een pakketbody. Dat is bijvoorbeeld handig als beginwaarden moeten worden toegekend, zoals in het vorige hoofdstuk bij DATA_BASE.DATA het geval was.

Het pakket COMPLEX kan nu bijvoorbeeld als volgt worden voltooid:

```
package body COMPLEX is
  function "+" (A,B : in GETAL) return GETAL is
    RESULTAAT: GETAL;
  begin
    RESULTAAT.REEEL_DEEL := A.REEEL_DEEL + B.REEEL_DEEL;
    RESULTAAT.IMAGINAIR_DEEL := A.IMAGINAIR_DEEL + B.IMAGINAIR_DEEL;
    return RESULTAAT;
  end "+";
  function "-" (A,B : in GETAL) return GETAL is
  begin
    return GETAL'(REEEL_DEEL     => A.REEEL_DEEL - B.REEEL DEEL,
                  IMAGINAIR_DEEL => A.IMAGINAIR_DEEL - B.IMAGINAIR_DEEL);
  end "-";
  function "*" (A,B : in GETAL) return GETAL is
    RESULTAAT : GETAL;
  begin
    RESULTAAT.REEEL_DEEL := (A.REEEL_DEEL * B.REEEL_DEEL) -
                            (A.IMAGINAIR_DEEL * B.IMAGINAIR_DEEL);
    RESULTAAT.IMAGINAIR_DEEL := (A.REEEL_DEEL * B.IMAGINAIR_DEEL) +
                                (A.IMAGINAIR_DEEL * B.REEEL_DEEL);
    return RESULTAAT;
  end "*";
end COMPLEX;
```

We gebruikten in bovenstaand voorbeeld twee manieren om een waarde via **return** te retourneren. Bij "+" en "*" gebruikten we een een lokale grootheid RESULTAAT en in de "-" functie gebruikten

we een geaggregeerde grootheid. Deze laatste methode wordt aanbevolen, tenzij dit te ingewikkeld wordt, zoals in de "*" functie. Initialisatie van grootheden in het pakket was hier niet nodig. Anders is dat bijvoorbeeld in het geval van een random generator (een generator van toevalsgetallen), waarbij een beginwaarde vereist is. Een pakketbody kan er dan zo uitzien:

```
package RANDOM is
  function GETAL return FLOAT range 0.0 .. 1.0;
end RANDOM;
package body RANDOM is
  START : INTEGER;
  function GETAL return FLOAT range 0.0 .. 1.0 is
    . . .
  end GETAL;
begin
  START := 1234567;
end RANDOM;
```

Hier wordt bij de verwerking van de body van het pakket RANDOM aan de variabele START een beginwaarde toegekend. Door weer betekenisvolle namen te gebruiken kunnen we nu RANDOM.GETAL gebruiken.

Voordat subprogramma's, die in het zichtbare deel van een pakketspecificatie werden gedeclareerd, ook daadwerkelijk kunnen worden aangeroepen, moeten zowel de specificatie als de body zijn verwerkt. Specificatie en body kunnen in de programmatekst afzonderlijk worden vermeld, zolang de specificatie maar eerst wordt gegeven. Vaak zullen we dan ook programma's van de volgende vorm tegenkomen:

```
procedure ALWEER_EEN_PROGRAMMA is
  package EERSTE is          -- specificatie van EERSTE
    . . .
  end EERSTE;
  procedure TWEEDE;          -- specificatie van TWEEDE
  procedure DERDE;           -- specificatie van DERDE
  package body EERSTE is     -- body van EERSTE
    . . .
  procedure TWEEDE is        -- body van TWEEDE
    . . .
  procedure DERDE is         -- body van DERDE
    . . .
begin
  . . .
end ALWEER_EEN_PROGRAMMA;
```

Zo worden eerst de interfaces van alle programma-eenheden gespecificeerd en worden hun bodies bij elkaar vermeld aan het eind van het declaratiegedeelte. Een nadeel van deze stijl is, dat het

hoofdprogramma behoorlijk lang kan worden en daardoor moeilijk leesbaar. Invoeren van subeenheden, zoals in het volgende voorbeeld kan zowel de programmatekst korter maken, alsook de uitwerking van de lokale eenheden verborgen houden:

```
procedure ZOVEELSTE_PROGRAMMA is
   package EERSTE is
      . . .
   end EERSTE;
   package body EERSTE  is separate;
   procedure    TWEEDE  is separate;
   procedure    DERDE   is separate;
begin
   . . .
end ZOVEELSTE_PROGRAMMA;
```

De bodies van de eenheden worden nu afzonderlijk gecompileerd, maar de specificaties zijn direct zichtbaar. In hoofdstuk 20 bespreken we het gebruik van deze methode verder.

13.2 Pakketten En Private Typen



Pakketten bevorderen de mogelijkheid tot het voorstellen van abstracte objecten. Het komt geregeld voor dat een programmeur een object wil creëren met bepaalde van buitenaf zichtbare logische eigenschappen, terwijl hij het interne mechanisme verborgen wil houden. Dit kan gebeuren door van private typen gebruik te maken. De definitie van zo'n type kan alleen maar in het zichtbare deel van een pakket voorkomen. Er bestaan twee categorieën:

- eenvoudige private typen
- beperkte private typen

In het geval van eenvoudige private typen is de enige informatie, die buiten het pakket beschikbaar is, de informatie, die wordt gegeven in het zichtbare gedeelte van dat pakket. Dat wil zeggen, dat wel de typenaam toegankelijk is, maar niet de waardenverzameling of de structuur. Wat betreft operaties zijn ook al weer alleen die operaties te gebruiken, die via de subprogramma's, die in het zichtbare deel van het pakket worden gedeclareerd, toepasbaar zijn, en verder nog de gelijkheid (=) en de ongelijkheid (/=). Dit alles geldt ook voor beperkte private typen, met dien verstande dat nu ook waardetoekenning en tests op gelijkheid of ongelijkheid buiten het pakket niet mogelijk zijn. Binnen het private gedeelte en de body van het pakket is de structuur van private typen natuurlijk wel zichtbaar en kan ernaar verwezen worden.

Bevat een pakket een definitie van een dergelijk 'privé' type, dan moet de specificatie ook een via het gereserveerde woord **private** aangegeven deel bevatten. Dit deel kan nog meer zaken bevatten, behalve de definitie van het **private** type, maar dit wordt in de regel niet beschouwd als een goede programmeerstijl. Tenslotte is het ook mogelijk dat een pakket zonder een **private** type toch een **private** gedeelte bevat.

Ook constanten kunnen van het type **private** zijn, maar objecten van dit type kunnen niet worden gedefinieerd, totdat het private deel is verwerkt. Stel, we willen bijvoorbeeld een pakket maken dat TOEGANGSCODEs uitgeeft en die ook controleert. De gecreëerde TOEGANGSCODE objecten willen we als beginwaarde bijvoorbeeld LEGE_CODE geven en deze waarde declareren we als een private constante:

```
package BEVEILIGING is
  type TOEGANGSCODE is private;
  LEGE_CODE : constant TOEGANGSCODE;
  function VRAAG return TOEGANGSCODE;
  function TOEGELATEN(P : in TOEGANGSCODE) return BOOLEAN;
private
  type TOEGANGSCODE is range 0 .. 7_000;
  LEGE_CODE : constant TOEGANGSCODE := 0;
end BEVEILIGING;
```

We maakten TOEGANGSCODE **private** en niet **limited private**, om gebruikers de mogelijkheid te geven elkaar toegangscodes toe te kennen. Waarden van het type TOEGANGSCODE kunnen echter niet worden benoemd, omdat zelfs de typedeclaratie niet zichtbaar is.

Wel kan LEGE_CODE buiten het pakket worden gebruikt, hoewel ook hier weer verborgen is hoe deze constante er eigenlijk uitziet. We kunnen bijvoorbeeld TOEGANGSCODE objecten een verstekwaarde (default waarde) geven:

```
    use BEVEILIGING;
    MIJN_TOEGANGSCODE : TOEGANGSCODE := LEGE_CODE;
```

Op deze manier kan de programmeur via private typen de wel en niet toegelaten operaties op geëxporteerde typen volledig beheersen. Vooral bij het creëren van de in de volgende paragraaf te behandelen datatypen is dit een zeer bruikbaar mechanisme.

13.3 Toepassingen Van Ada Pakketten



Ada is een zeer krachtige algemeen bruikbare taal, maar daarom is het nog niet onmogelijk eigenschappen van de taal te misbruiken

- het is heel goed mogelijk onleesbare en ongestructureerde Ada programma's te schrijven. Om dit te vermijden moeten de mogelijkheden van de taal doelmatig worden toegepast. Zeker bij het gebruik van pakketten is dat het geval: zij vormen een essentieel onderdeel van elk in Ada geschreven systeem. Ada pakketten moeten logisch gezien klein zijn: zij moeten een compacte bouwsteen van logisch samenhangende grootheden opleveren. De toepassingen van pakketten kunnen in vier categorieën worden verdeeld, namelijk:

- van een naam voorziene declaraties
- groepen samenhangende programma-eenheden
- abstracte datatypen
- abstracte automaten

Elk van deze categorieën kan nader worden gekarakteriseerd met behulp van de grootheden die gewoonlijk door pakketten uit die categorie worden geëxporteerd:

- *Van een naam voorziene verzameling declaraties*
  Exporteren objecten en typen
  Exporteren geen andere programma-eenheden
- *Groepen samenhangende programma-eenheden*
  Exporteren geen objecten en typen
  Exporteren andere programma-eenheden
- *Abstracte datatypen*
  Exporteren objecten en typen
  Exporteren andere programma-eenheden
  Houden geen toestandsinformatie bij
- *Abstracte automaten*
  Exporteren objecten en typen
  Exporteren andere programma-eenheden
  Houden toestandsinformatie bij

Hierboven zijn de toepassingen in hun zuivere vorm weergegeven; in de praktijk zullen vaak vermengingen optreden. We zullen nu voorbeelden geven van elk van deze toepassingsgebieden.

Van een naam voorziene verzamelingen declaraties

Het tesamen brengen van logisch samenhangende objecten en typen is een van de eenvoudigste toepassingen van het pakket. Door gemeenschappelijke gegevens, objecten en typen slechts één maal te noemen op één bepaalde plaats wordt in ieder geval de onderhoudbaarheid van de programmatuur bevorderd. Andere programma-eenheden kunnen van de in het pakket opgenomen definities gebruik maken; als een definitie moet worden aangepast, dan hoeft dat maar op één plaats te gebeuren.

We geven een voorbeeld: in een systeem voor plaatsbepaling op aarde is een model van de wereldbol nodig. In dit model zijn een aantal constanten opgenomen die door de andere programma-eenheden kunnen worden gebruikt. We kunnen hier het volgende pakket voor formuleren:

```
package METRISCHE_AARDE_CONSTANTEN is
  EQUATOR_STRAAL              : constant := 6_378.145;     -- km
  ZWAARTEKRACHT_CONSTANTE : constant := 3.986_012e5;   -- km**3/sec**2
  SNELHEIDSEENHEID            : constant := 7.905_368_28;  -- km/sec
  TIJDSEENHEID                : constant := 806.811_874_4; -- sec
end METRISCHE_AARDE_CONSTANTEN;
```

Er worden in dit voorbeeld in de specificatie geen andere programma-eenheden gedefinieerd, dus mag de pakketbody worden weggelaten. Nog veiliger zou zijn, om typen die kilometers en seconden definiëren te exporteren en vervolgens de getallen als constanten van de bijbehorende typen te declareren. Als we dat doen, kan de gebruiker de constanten niet met de verkeerde meeteenheden gebruiken. Voorgedefinieerde typen, zoals FLOAT, moeten bij voorkeur vermeden worden om het systeem overdraagbaar (toepasbaar op verschillende apparatuur) te maken. In dit voorbeeld gebruikten we numerieke constanten, die automatisch van het type universal_real zijn, in plaats van FLOAT constanten.

Vaak is het zinvol om een aantal logisch samenhangende grootheden tesamen te brengen in één pakket. We geven weer een voorbeeld. Een systeem, waarin veel met een kalender moet worden gewerkt, kan dagen, maanden en jaren in één pakket opnemen:

```
package KALENDER_GEGEVENS is
   type DAG_NAAM    is (MAANDAG,DINSDAG,WOENSDAG,
                        DONDERDAG,VRIJDAG,ZATERDAG,
                        ZONDAG);
   type DAG_WAARDE  is range 1 .. 31;
   type MAAND_NAAM  is (JANUARI,FEBRUARI,MAART,APRIL,
                        MEI,JUNI,JULI,AUGUSTUS,SEPTEMBER,
                        OKTOBER,NOVEMBER,DECEMBER);
   type JAAR_WAARDE is range 0 .. INTEGER'LAST;
end KALENDER_GEGEVENS;
```

Ook hier is een pakketbody weer niet noodzakelijk.

Nog wat opmerkingen over programmeerstijl: we gebruikten in de voorbeelden betekenisvolle namen, zelfs als die nogal lang uitvallen. Dit bevordert de begrijpelijkheid van het geheel. Ook hielden we de pakketten klein, want grote pakketspecificaties zijn lastig leesbaar en moeilijk te begrijpen. Zonodig kan gebruik worden gemaakt van *subpakketten* (pakketten binnen een pakket). Zelfs als pakketten louter gebruikt worden als verzamelingen declaraties, dan nog dienen zij niet beschouwd te worden als COMMON gebieden, zoals in FORTRAN, of als COMPOOLs, zoals in JOVIAL. Dit zou

indruisen tegen de opzet van Ada en een deel van de kracht van de taal zou zo verloren gaan.

Groepen samenhangende programma-eenheden

In de voorgaande voorbeelden rangschikten we logisch samenhangende gegevens binnen één pakket. Ditzelfde kan met programma-eenheden, zoals subprogramma's, taken en zelfs andere pakketten. Ada heeft bijvoorbeeld geen voorgedefinieerde trigonometrische functies, zoals sinus en cosinus, en die zouden we in een pakket kunnen definiëren. (Dit is maar een voorbeeld. In werkelijkheid zou de programmeur gebruik kunnen maken van bibliotheekfuncties.) Het pakket zou nu als volgt kunnen worden gespecificeerd:

```
package TRANSCENDENTE_FUNCTIES is
   type RADIALEN  is digits 5;
   type RESULTAAT is digits 7;
   function COS(HOEK : in RADIALEN) return RESULTAAT;
   function SIN (HOEK : in RADIALEN) return RESULTAAT;
   function TAN(HOEK : in RADIALEN) return RESULTAAT;
end TRANSCENDENTE_FUNCTIES;
```

Ook hier gebruikten we niet het voorgedefinieerde type FLOAT, maar maakten we RADIALEN en RESULTAAT afzonderlijke typen, elk met hun eigen precisie. In een produktie-omgeving zou de bruikbaarheid van het pakket nog kunnen worden vergroot door een generiek gedeelte toe te voegen, zodat het pakket zou kunnen werken met verschillende door de gebruiker te definiëren precisies. Dit zullen we in het volgende hoofdstuk dan ook doen.

Hier is wel een pakketbody nodig, omdat nu in de specificatie behalve objecten, constanten en typen, ook functies worden gebruikt. De gebruiker hoeft deze body vanzelfsprekend niet te kunnen zien: welke algoritmen worden gebruikt voor het berekenen van de functiewaarden is voor hem niet van belang. Het zou in dit geval verstandig zijn als de ontwikkelaar van de pakketbody deze afzonderlijk zou compileren: mochten de algoritmen moeten worden aangepast (bijvoorbeeld omdat een efficiëntere algoritme werd ontwikkeld), dan heeft deze aanpassing geen effect op de programmatuur, die van het pakket gebruik maakt. We laten hier een uitwerking zien van een pakketbody voor de berekening van de functies via reeksontwikkeling. Het gaat hier niet om een zo efficiënt mogelijke algoritme (dat laten we over aan de numerici), maar slechts om een voorbeeld van het gebruikt van Ada:

```
package body TRANSCENDENTE_FUNCTIES is
  REEKS_LENGTE : constant := 5;
  function ONEVEN(INDEX : in INTEGER) return BOOLEAN is
    -- retourneer de waarde TRUE als INDEX oneven is
  begin
    return ((INDEX mod 2) /= 0);
  end ONEVEN;
  function FACULTEIT(WAARDE : in INTEGER) return INTEGER is
    -- bepaal WAARDE! met behulp van een recursieve functie
  begin
    if WAARDE = 1 then
      return 1;
    else
      return (WAARDE * FACULTEIT(WAARDE - 1));
    end if;
  end FACULTEIT;
  function TERM(HOEK  : in RADIALEN,
                MACHT : in INTEGER,
                GETAL : in INTEGER) return RESULTAAT is
    -- bereken een term van de reeks
  begin
    if ONEVEN(GETAL) then
      return RESULTAAT(-((HOEK)**MACHT)/(RADIALEN(FACULTEIT(MACHT))));
    else
      return RESULTAAT(+((HOEK)**MACHT)/(RADIALEN(FACULTEIT(MACHT))));
    end if;
  end TERM;
  function COS(HOEK : in RADIALEN) return RESULTAAT is
    -- bereken de cosinus van de HOEK
    ANTWOORD : RESULTAAT;
    MACHT    : INTEGER;
  begin
    ANTWOORD := 1.0;
    for I in reverse 1 .. REEKS_LENGTE
      loop
        MACHT := I*2;
        ANTWOORD := ANTWOORD + TERM(HOEK,MACHT.GETAL);
      end loop;
    return ANTWOORD;
  end COS;
  function SIN(HOEK : in RADIALEN) return RESULTAAT is
    -- bereken de sinus van de HOEK
    ANTWOORD : RESULTAAT;
    MACHT    : INTEGER;
  begin
    ANTWOORD := RESULTAAT(HOEK);  -- conversie naar zeven cijfers
    for I in reverse 1 .. REEKS_LENGTE
      loop
        MACHT := (I*2) + 1;
        ANTWOORD := ANTWOORD + TERM(HOEK,MACHT,GETAL);
      end loop;
    return ANTWOORD;
  end SIN;
  function TAN(HOEK : in RADIALEN) return RESULTAAT is
    -- bereken de tangens van de HOEK
  begin
    return (SIN(HOEK)/COS(HOEK));
  end TAN;
end TRANSCENDENTE FUNCTIES;
```

De constante REEKS_LENGTE werd hier op een speciale manier gebruikt: door deze grootheid als een constante te declareren kan het pakket later gemakkelijker worden gewijzigd. In de body werden geen instructies voor initialisatie van variabelen opgenomen; dit gebeurt in drie lokale subprogramma's. Deze functies: ONEVEN, FACULTEIT en TERM zijn niet van buiten het pakket te benaderen; zij zijn in het pakket 'verborgen'. De functies SIN en COS doorlopen de termen van de reeks van klein naar groot, om de afrondingsfout zo klein mogelijk te houden. De functie TAN wordt met behulp van de SINus en COSinus berekend.

Ook een pakket voor grafische toepassing kan als voorbeeld dienen voor de toepassing van pakketten in Ada als verzamelingen van subprogramma's. In een dergelijk pakket zullen in ieder geval transformaties, zoals ROTEER, SCHAAL en TRANSLEER (verschuif), voorkomen en de pakketspecificatie zou er daarom als volgt kunnen uitzien:

```
package TWEE_DIM_TRANSFORMATIE is
  type COORDINAAT is record
                          X : FLOAT;
                          Y : FLOAT;
                        end record;
  procedure ROTEER     (PUNT : in out COORDINAAT;
                         HOEK : in     FLOAT);
  procedure SCHAAL      PUNT : in out COORDINAAT;
                         X,Y  : in     FLOAT);
  procedure TRANSLEER  (PUNT : in out COORDINAAT;
                         X,Y  : in     FLOAT);
end TWEE_DIM_TRANSFORMATIE;
```

We zijn hier afgeweken van onze stelregel, nooit voorgedefinieerde typen, zoals FLOAT te gebruiken, maar in dit geval deden we dit om de oplossing eenvoudig te houden en ook om nog wat toepassingen van typetransformaties te laten zien.

In ons pakket zullen we hulpmiddelen uit het pakket TRANSCENDENTE_FUNCTIES nodig hebben, verwijzigingen daarnaar kunnen we echter verbergen in de body van TWEE_DIM_TRANSFORMATIE. Verder gebruiken we hier de use-clausule, om de namen kort te houden; van dubbelzinnigheid kan hier nauwelijks sprake zijn omdat de pakketspecificatie erg klein is. Een body zou er nu als volgt kunnen uitzien:

```
with TRANSCENDENTE_FUNCTIES;
use TRANSCENDENTE_FUNCTIES;
package body TWEE_DIM_TRANSFORMATIE is
  procedure ROTEER(PUNT : in out COORDINAAT;
                   HOEK : in     FLOAT) is
    -- roteer PUNT over HOEK radialen rond de oorsprong
  TEMP : COORDINAAT := PUNT;
```

```
  begin
    PUNT.X :=  (PUNT.X * FLOAT(COS(RADIALEN(HOEK)))) +
               (PUNT.Y * FLOAT(SIN(RADIALEN(HOEK))));
    PUNT.Y := - (PUNT.X * FLOAT(SIN(RADIALEN(HOEK)))) +
               (PUNT.Y * FLOAT(COS(RADIALEN(HOEK))));
  end ROTEER;
  procedure SCHAAL(PUNT : in out COORDINAAT;
                   X,Y  : in     FLOAT) is
    -- vermenigvuldig PUNT met schaalfactoren X en Y
  begin
    PUNT.X := PUNT.X * Y;
    PUNT.Y := PUNT.Y * Y;
  end SCHAAL;
  procedure TRANSLEER(PUNT : in out COORDINAAT;
                      X,Y  : in     FLOAT) is
    --transleer het PUNT over afstanden X en Y
  begin
    PUNT.X := PUNT.X + X;
    PUNT.Y := PUNT.Y + Y;
  end TRANSLEER;
end TWEE_DIM_TRANSFORMATIE;
```

Ook hier is weer geen pakket-initialisatie nodig.

We hebben nu een paar toepassingen gezien van pakketten als verzamelingen van subprogramma's. Pakketten kunnen ook worden gebruikt voor het bijeenbrengen van andere pakketten of van taken. Een voorbeeld daarvan zagen we al in het tweede ontwerpprobleem.

Abstracte datatypen

Ada bezit een groot aantal elementaire datatypen (zie nog eens hoofdstuk 8). Toch dekt deze verzameling natuurlijk niet alle mogelijke behoeften en daarom biedt de taal een mechanisme, waarmee de gebruiker zelf een abstract datatype kan creëren. Dit datatype wordt zo ingekapseld, dat de wijze van gebruik door de taal dwingend wordt opgelegd. We lieten al zien, dat dit mechanisme gebruik maakt van pakketten en van private typen. Een goede stelregel is, om per pakket maar één type, of een paar sterk samenhangende typen, te exporteren. Een pakketspecificatie bevat dan maar één **private** of **limited private** type, de subprogramma's, die de operaties op dat type definiëren, en het private specificatiegedeelte.

Het pakket COMPLEX was al een voorbeeld van een abstract datatype. Deze abstractie kan echter niet bindend worden opgelegd, omdat niet voor het type **private** werd gekozen, zodat de structuur zichtbaar blijft. Een gebruiker van het pakket kan daarom inbreuk doen op de regels en bijvoorbeeld het reëel en imaginair deel van een getal bij elkaar optellen. De taal zou een dergelijke fout niet kunnen signaleren.

Om een logische abstractie dwingend te maken moeten we private typen gebruiken. In het geval van het pakket COMPLEX zouden we het type GETAL **private** kunnen declareren en de uitwerking van dit type in een **private** gedeelte kunnen onderbrengen. Nu zou de structuur niet meer direct kunnen worden benaderd en dat betekent dat bijvoorbeeld de operaties HAAL_REEEL_DEEL en GEEF_REEEL_DEEL zouden moeten worden toegevoegd. De volledige specificatie van COMPLEX laten we als oefening aan de lezer over.

Ada kent geen standaardwachtrij datastructuren, zoals bijvoorbeeld een FIFO ('first-in-first-out') buffer. We gaan daarom een FIFO_RIJ datatype maken, dat vervolgens als een elementair datatype kan worden behandeld. Ook willen we daarbij wachtrijen van verschillende lengtes kunnen definiëren. Ter vereenvoudiging van het probleem definiëren we alleen RIJen met INTEGER elementen, hoewel het ook niet zo moeilijk zou zijn een generieke RIJ te creëren, geschikt voor elk type element (zie het volgende hoofdstuk). We definiëren nu het abstracte datatype RIJ in de volgende pakketspecificatie:

```
package FIFO is
  type RIJ(LENGTE : NATURAL) is limited private;
  procedure SCHOON(BUFFER : out RIJ);
  procedure HAAL   (WAARDE : out INTEGER; UIT : in out RIJ);
  procedure PLAATS (WAARDE : in  INTEGER; IN  : in out RIJ);
  RIJ_VOL  : exception;
  RIJ_LEEG : exception;
private
  type LIJST is array (INTEGER range <>) of INTEGER;
  type RIJ(LENGTE : NATURAL) is
    record
      ELEMENTEN : LIJST(0 .. LENGTE);
      VOORSTE   : INTEGER;
      ACHTERSTE : INTEGER;
    end record;
end FIFO;
```

We gebruikten een record discriminant in de private typedeclaratie om de onderling samenhangende elementen van de RIJ aan te geven. De toegelaten operaties op dit abstracte datatype zijn SCHOON, HAAL en PLAATS. RIJ werd als **limited private** gedeclareerd om te voorkomen, dat de gebruiker bijvoorbeeld RIJ objecten bij elkaar optelt of op gelijkheid test. Er worden ook twee zogenaamde excepties geëxporteerd, RIJ_VOL en RIJ_LEEG, om te reageren op een poging om een element toe te voegen aan een vol buffer, of een element te nemen uit een leeg buffer. In het volgende hoofdstuk zullen we laten zien hoe de FIFO.RIJ algemener bruikbaar kan worden gemaakt door een generiek gedeelte toe te voegen. Dan is het mogelijk WAARDEn van verschillende typen via een parameter door te geven. Ook hier hoeft de gebruiker weer niet te kunnen zien hoe de operaties zijn geprogrammeerd.

Er bestaan verschillende oplossingen voor het maken van FIFO wachtrijen, zoals via dynamische verbonden lijsten of via eenvoudige arrays. Hier kiezen wij voor een oplossing met een cirkelvormig buffer. In dat geval moet er één RIJ-element meer zijn dan de totale lengte van de wachtrij en dit wordt mogelijk via de variabelen VOORSTE en ACHTERSTE. De hele oplossing blijft voor de gebruiker verborgen in het private gedeelte en de gebruiker kan objecten declareren, zoals:

```
MIJN_BUFFER : FIFO.RIJ(70);              -- rij ter lengte 70
UW_BUFFER   : FIFO.RIJ(LENGTE => 32);    -- rij ter lengte 32
```

Door het gebruik van de private record discriminant kan de gebruiker rijen van verschillende lengte declareren.

De pakketbody kan nu bijvoorbeeld als volgt worden geprogrammeerd:

```
package body FIFO is
  procedure SCHOON(BUFFER : out RIJ) is
  begin
    VOORSTE   : BUFFER.ELEMENTEN'LAST;
    ACHTERSTE : BUFFER.ELEMENTEN'LAST;
  end SCHOON;
  procedure HAAL(WAARDE : out INTEGER; UIT : in out RIJ) is
    -- haal een WAARDE uit de RIJ
  begin
    if UIT.VOORSTE = UIT.ACHTERSTE then
      raise RIJ_LEEG;
    else
      if UIT.VOORSTE = UIT.ELEMENTEN'LAST then
        UIT.VOORSTE := UIT.ELEMENTEN'EERSTE;
      else
        UIT.VOORSTE := UIT.VOORSTE + 1;
      end if;
      WAARDE := UIT.ELEMENTEN(UIT.VOORSTE);
    end if;
  end HAAL;
  procedure PLAATS(WAARDE : in INTEGER; IN : in out RIJ) is
    -- plaats een WAARDE in de RIJ
  begin
    if IN.ACHTERSTE = IN.ELEMENTEN'LAST then
      IN.ACHTERSTE := IN.ELEMENTEN'FIRST
    else
      IN.ACHTERSTE := IN.ACHTERSTE + 1;
    end if;
    if IN.VOORSTE = IN.ACHTERSTE then
      raise RIJ_VOL;
    else
      IN.ELEMENTEN(IN.ACHTERSTE) := WAARDE;
    end if;
  end PLAATS;
end FIFO;
```

Een gebruiker van dit pakket moet eerst de gedeclareerde RIJen SCHOON maken. Door het uitvoeren van deze laatste operatie gaat zowel VOORSTE als ACHTERSTE verwijzen naar het laatste element van de RIJ. Als de operatie HAAL wordt uitgevoerd, dan wordt eerste gecontroleerd op RIJ_LEEG en is dit niet het geval, dan wordt de verwijzing VOORSTE één plaats opgeschoven. Bij het uitvoeren van de operatie PLAATS wordt eerst de verwijzing ACHTERSTE opgeschoven en vervolgens gecontroleerd of de rij vol is. Als RIJ_VOL geldt dan wordt de exceptievlag gehesen. Omdat VOORSTE behouden blijft kan het systeem zich vervolgens zonder probleem van de fout herstellen.

We bekijken nog een andere toepassing van Ada pakketten als abstracte datatypen. In een procesbesturingssysteem kunnen bijvoorbeeld opslagtanks behandeld moeten worden. In Ada komt natuurlijk geen elementair datatype TANK voor, maar alweer met behulp van private typen kan deze abstractie worden gecreëerd. De gebruiker van het pakket krijgt bijvoorbeeld het volgende te zien:

```
package OPSLAG is
   type TANK is limited private;
   type PERCENTAGE is delta 0.01 range 0.0 .. 100.0;
   procedure VOEG_TOE(HOEVEELHEID : in PERCENTAGE; AAN : in out TANK);
   function  NIVEAU(PLAATS       : in TANK) return PERCENTAGE;
   procedure VERWIJDER(HOEVEELHEID: in PERCENTAGE; UIT : in out TANK);
   IS_LEEG : exception;
   IS_VOL  : exception;
private
   type TANK is new PERCENTAGE;
end OPSLAG;
```

Door de inkapseling kan de abstractie niet doorbroken worden en is het bijvoorbeeld niet mogelijk twee TANK objecten bij elkaar op te tellen. Ook maakt de **limited private** declaratie het onmogelijk, de waarde van het ene TANK object aan een ander toe te kennen, hetgeen immers ook in werkelijkheid fysiek onmogelijk is. In plaats daarvan moet uit het ene object VERWIJDERd worden en geldt de operatie VOEG_TOE voor het andere object. Verdere beveiliging wordt verzorgd door de excepties. Een exceptie-situatie moet worden gesignaleerd, *voordat* een poging wordt gedaan een niet toegelaten actie uit te voeren, zoals bijvoorbeeld een poging tot bijvullen van een volle tank. Hoe het mogelijk is een fout te signaleren voordat deze daadwerkelijk wordt begaan, zullen we laten zien na de volgende uitwerking:

```
package body OPSLAG is
  procedure VOEG_TOE(HOEVEELHEID : in PERCENTAGE; AAN : in out TANK) is
    -- voeg HOEVEELHEID toe aan TANK object AAN
  begin
    AAN := AAN + TANK(HOEVEELHEID);
  exception
    when CONSTRAINT_ERROR => raise IS_VOL;
  end VOEG_TOE;
  function NIVEAU(PLAATS : in TANK) return PERCENTAGE is
    -- retourneer NIVEAU van TANK object PLAATS
  begin
    return PERCENTAGE(PLAATS);
  end NIVEAU;
  procedure VERWIJDER(HOEVEELHEID : out PERCENTAGE; UIT : in out TANK) is
    -- neem HOEVEELHEID weg uit TANK object UIT
  begin
    UIT := UIT - TANK(HOEVEELHEID);
  exception
    when CONSTRAINT_ERROR => raise IS_LEEG;
  end VERWIJDER;
end OPSLAG
```

Hoewel het hier om vrij simpele algoritmen gaat, worden zij toch van de gebruiker afgeschermd. Ook de toegepaste typetransformaties zijn voor de gebruiker niet zichtbaar. Verder is te zien hoe de voorgedefinieerde exceptie CONSTRAINT_ERROR wordt afgevangen en als een van de door de gebruiker gedefinieerde excepties (IS_VOL of IS_LEEG) wordt doorgegeven. De fout wordt hierbij gesignaleerd, voordat enige schade kan worden aangericht; toevoegen aan een bijna volle tank leidt tot een poging tot toekenning van de nieuwe waarde, maar als deze te groot is ontstaat de exceptie CONSTRAINT_ERROR en wordt de waardetoekenning niet uitgevoerd. Dit betekent dat het TANK object de waarde behoudt die het had voordat de exceptie optrad en het systeem herstelt zich op een eenvoudige manier van de fout.

Abstracte automaten

Een *automaat* is een grootheid, die zich in een aantal welomschreven toestanden kan bevinden en waarbij tevens is beschreven welke invoer een toestand naar een volgende toestand doet overgaan en welke uitvoer daarbij gegenereerd wordt.

Als een pakket wordt gebruikt als een abstracte automaat, dan exporteert het pakket meestal geen typen of objecten. Een dergelijk pakket lijkt veel op een abstract datatype of een verzameling programma-eenheden. Het grote verschil is echter, dat een pakket als automaat informatie over de toestand van de automaat registreert. Het pakket TRANSCENDENTE_FUNCTIES bijvoorbeeld bestond uit een aantal subprogramma's, maar het aanroepen van een subprogramma veranderde de toestand van het geheel niet. De subprogramma's waren onderling onafhankelijk en het resultaat van het ene

subprogramma beïnvloedde geen der andere. Worden binnen een automaat operaties toegepast, dan verandert de toestand wel degelijk. We zouden bijvoorbeeld een OVEN als volgt aan de gebruiker kunnen voorstellen:

```
package OVEN is
  function IS_AAN return BOOLEAN;
  procedure STEL_IN(TEMPERATUUR : in FLOAT);
  procedure DOE_UIT;
  function TEMPERATUUR_IS return FLOAT;
  OVERVERHIT : exception;
end OVEN;
```

De gebruiker van het pakket kan alleen maar de vier subprogramma's toepassen om de toestand van de OVEN te veranderen of om de huidige TEMPERATUUR vast te stellen. De volgorde, waarin de subprogramma's worden aangeroepen, heeft hier wel degelijk invloed, omdat gegevens binnen het pakket behouden blijven. Het pakket definieert maar één object en er worden geen typen of objecten geëxporteerd. Verder worden er in de pakketspecificatie een aantal subprogramma's gedefinieerd en dus zal ook een volledige pakketbody moeten worden uitgewerkt.

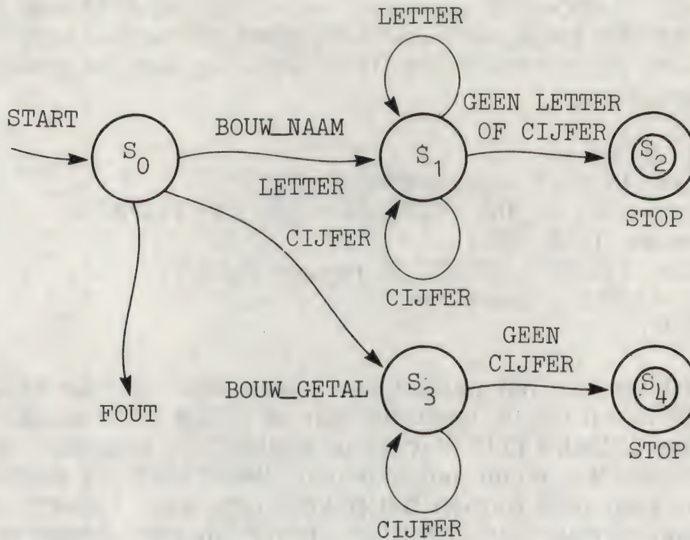
Een bekend voorbeeld van een automaat is een zogenaamde *lexical analyzer*, een automaat die rijen symbolen analyseert en daarin elementen uit een bepaalde (programmeer)taal kan herkennen. Zowel binnen compilers als bijvoorbeeld in een communicatiesysteem worden dergelijke automaten toegepast. Binnen een compiler willen we bijvoorbeeld namen en getallen kunnen herkennen. Een toestandsdiagram voor een automaat die dat kan is in figuur 13-2 weergegeven.

De automaat begint in de toestand START. Zodra het eerste symbool binnenkomt gaat een automaat over in de BOUW_NAAM toestand als dit symbool een letter is en in de BOUW_GETAL toestand als het geen letter is. De automaat blijft in de toestand BOUW_NAAM zolang letters blijven binnenkomen en in de toestand BOUW_GETAL zolang cijfers binnenkomen. Zodra een afwijkend symbool worden gelezen, gaat de automaat over in de toestand STOP; de grootheid is nu herkend.

De interface voor deze automaat kan als volgt worden beschreven:

```
package LEXICAL_ANALYZER is
  procedure BRENG_IN_START_TOESTAND;
  procedure LEES_SYMBOOL(C : in CHARACTER);
  NAAM_GEACCEPTEERD       : exception;
  NIET_TOEGELATEN_TEKEN : exception;
  AUTOMAAT_IS_GESTOPT     : exception;
  GETAL_GEACCEPTEERD      : exception;
end LEXICAL_ANALYZER;
```

We definieerden alle eindtoestanden als excepties; de gebruiker van het pakket kan op die manier doorgaan met sturen van symbolen,



Figuur 13-2 Automaat voor LEXICAL_ANALYZER

zonder dat hij zelf expliciet hoeft te testen of een grootheid geaccepteerd is. Het zou natuurlijk best mogelijk zijn, informatie over het al dan niet geaccepteerd zijn van een symbolenrij via een output parameter in LEES_SYMBOOL door te geven. Als het eerste teken noch een letter, noch een cijfer is, dan ontstaat de NIET_TOEGELATEN_TEKEN exceptie. Verder voegden we een operatie BRENG_IN_START_TOESTAND toe om de automaat te kunnen hetstarten en een AUTOMAAT_IS_GESTOPT toestand, voor het geval dat de gebruiker iets probeert te doen met de automaat in STOP-toestand.

De automaat kan nu als volgt worden uitgeschreven:

```
package body LEXICAL_ANALYZER is
  type TOESTAND is (START,BOUW_NAAM,BOUW_GETAL,STOP);
  HUIDIGE_TOESTAND : TOESTAND := START;
  subtype ALPHA is CHARACTER range 'A' .. 'Z';
  subtype CIJFER is CHARACTER range '0' .. '9';
  procedure BRENG_IN_START_TOESTAND is
    --initialiseer de automaat
  begin
    HUIDIGE_TOESTAND := START;
  end BRENG_IN_START_TOESTAND;
```

```
    procedure LEES_SYMBOOL(C : in  CHARACTER) is
      -- lees een verzonnen symbool
    begin
      case HUIDIGE TOESTAND is
        when START        => if (C in ALPHA) then
                                HUIDIGE_TOESTAND := BOUW_NAAM;
                              elsif (C in CIJFER) then
                                HUIDIGE_TOESTAND := BOUW_GETAL;
                              else
                                raise NIET_TOEGELATEN_TEKEN;
                              end if;
        when BOUW_NAAM  => if (C in ALPHA) or (C in DIGIT) then
                                null;
                              else
                                HUIDIGE_TOESTAND := STOP;
                                raise NAAM_GEACCEPTEERD;
                              end if;
        when BOUW_GETAL => if (C in CIJFER) then
                                null;
                              else
                                HUIDIGE_TOESTAND := STOP;
                                raise GETAL_GEACCEPTEERD;
                              end if;
        when STOP          => raise AUTOMAAT_IS_GESTOPT;
      end case;
    end LEES_SYMBOOL;
end LEXICAL_ANALYZER;
```

En hiermee is onze eenvoudige lexical analyzer gereed. De automaat kan gemakkelijk uitgebreid worden, omdat we TOESTAND als een enumeratietype definieerden.

De kracht van het pakketmechanisme in Ada zit in de mogelijkheid, grootheden in te kapselen, zodanig dat het niet mogelijk is om van de eenmaal vastgelegde abstracties af te wijken. In het volgende hoofdstuk behandelen we nog een pakketmogelijkheid in verband met generieke programma-eenheden.

Oefeningen

1. Formuleer een pakketspecificatie COMPLEX, die een grootheid GETAL van het type **private** exporteert. Vul het pakket aan met een body, waarin operaties voor optellen, aftrekken, vermenigvuldiging, delen, het toekennen en opvragen van waarden en de bepaling van de modulus en het argument zijn opgenomen.

2. Maak een pakket METRISCHE_ENGELSE_CONVERSIE, dat de typen LITER en GALLON en de typen CENTIMETER en INCH exporteert, tesamen met de constanten met geschikte waarden van het type universal_real, voor de berekening van de conversies.

3. Schrijf een pakketspecificatie voor de conversie van de normale dag-maand-jaar notatie naar de overeenkomstige Juliaanse datum. (De Juliaanse notatie bestaat uit een jaarnummer en het nummer van de dag in dit jaar. 128 83 is bijvoorbeeld de 128-ste dag van het jaar 1983.)

4. Formuleer de functie FACULTEIT in het pakket TRANSCENDENTE_FUNCTIES opnieuw, maar nu zonder gebruik te maken van recursie.

5. Schrijf een pakketspecificatie voor driedimensionale transformaties (schaling, rotatie en translatie).

*6. Schrijf de body van FIFO opnieuw en maak daarbij deze keer gebruik van access typen om de RIJ te implementeren.

*7. Pas de LEXICAL_ANALYZER zo aan, dat een exceptie optreedt, telkens als een niet toegelaten teken worden gelezen.

*8. Wijzig de LEXICAL_ANALYZER vervolgens zo, dat symbolen als ':=', '+' en '-' worden herkend.

14 GENERIEKE PROGRAMMA-EENHEDEN

Als een softwaresysteem wordt onderverdeeld in modulen, dan blijken er vaak een aantal subprogramma's en pakketten te zijn, die voor vrijwel hetzelfde doel zijn ontworpen. Vaak is bijvoorbeeld een sorteerprocedure nodig, om waarden van een bepaald type te sorteren, voordat een overzichtsrapport wordt geproduceerd (denk aan het database opvraagprobleem uit de hoofdstukken 9 en 12). De sorteerbewerking kan gemakkelijk worden uitgevoerd met behulp van een Ada subprogramma, zelfs voor variabele array-lengtes. Hiervoor zou de volgende subprogrammaspecificatie kunnen worden gebruikt:

```
type INTEGER_ARRAY is array (NATURAL range <>) of INTEGER;
procedure SORT(MIJN_ARRAY : in out INTEGER_ARRAY);
```

Voor het sorteren van een array met elementen van een ander type is dan weer een ander subprogramma nodig. Het type van elk object moet immers in een sterk getypeerde taal als Ada tijdens de compilatie bekend zijn. We zijn dus wel gedwongen weer een declaratie te schrijven:

```
type REAL_ARRAY is array (NATURAL range <>) of FLOAT;
procedure SORT(MIJN_ARRAY : in out REAL_ARRAY);
```

Dit is duidelijk een ongewenste situatie; voor elk nieuw type moeten we een nieuw pakket creëren, terwijl de sorteeralgoritme precies hetzelfde blijft. De complexiteit van het systeem neemt hierdoor toe en criteria als wijzigbaarheid, betrouwbaarheid en begrijpelijkheid worden nadelig beïnvloed. We hebben behoefte aan 'blauwdrukken' voor programma-eenheden, die maar één keer hoeven te worden geformuleerd en waarin de specifieke behoeften op het moment van van de vertaling kunnen worden ingevuld. Ada beschikt over een algemeen en zeer krachtig hulpmiddel om dit voor elkaar te krijgen, namelijk de generieke programma-eenheden.

14.1 Generieke Programma-Eenheden: Vorm

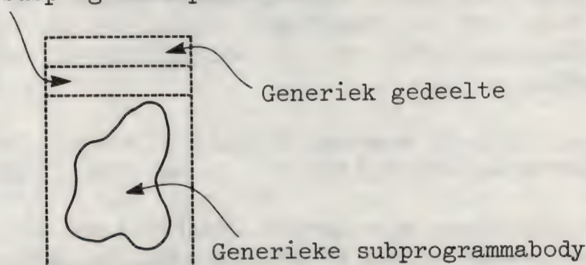


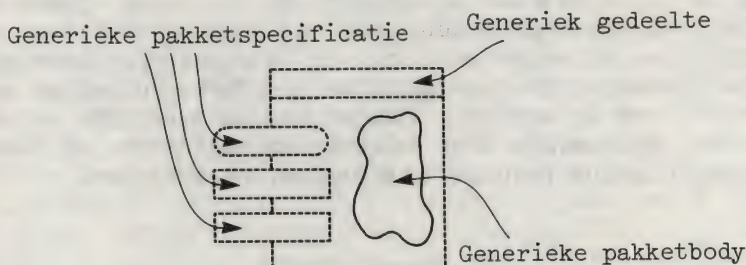
In Ada kunnen generieke pakketten en subprogramma's worden gedefinieerd. We zullen later zien, dat ook generieke taken mogelijk zijn; daartoe moet de taak worden ingebed in een pakket. Generieke programma-eenheden zijn blauwdrukken voor programma-eenheden, tesamen met generieke parameters, die het mogelijk maken om de eenheid voor een specifiek gebruik aan te passen.

Juist omdat generieke eenheden alleen maar blauwdrukken zijn, kunnen zij niet worden uitgevoerd; zij kunnen dus niet direct worden gebruikt. Eerst moet er een specifieke verschijningsvorm van de generieke eenheid worden gecreëerd en pas dan kan het subprogramma of pakket als een gewone programma-eenheid worden gebruikt. Generieke eenheden verhouden zich tot subprogramma's op ongeveer dezelfde manier als typen zich verhouden tot objecten. In figuur 14-1 laten we zien hoe onze grafische voorstelling van subprogramma's en pakketten in Ada kan worden aangepast voor generieke eenheden.

De 'generic' definitie

Een generieke programma-eenheid kan worden gecreëerd, door een pakket- of subprogrammaspecificatie te nemen en daar een *generiek*





Figuur 14-1 Symbolische weergave van generieke programma-eenheden

gedeelte voor te zetten. Dit generiek gedeelte definieert tevens alle generieke parameters (als deze tenminste worden gebruikt). Een voorbeeld: we formuleren een subprogramma, dat twee elementen van het type INTEGER onderling verwisselt:

```
procedure WISSEL_INTEGER(EERSTE,TWEEDE : in out INTEGER) is
  TIJDELIJK : INTEGER;
begin
  TIJDELIJK := EERSTE;
  EERSTE    := TWEEDE;
  TWEEDE    := TIJDELIJK;
end WISSEL_INTEGER;
```

Als we nu vervolgens grootheden van een ander type willen verwisselen, dan is het niet nodig telkens een nieuw subprogramma te maken. We kunnen het subprogramma algemeen toepasbaar maken door er een generiek gedeelte voor te zetten:

```
generic
  type ELEMENT is private;
procedure WISSEL(EERSTE,TWEEDE : in out ELEMENT);
```

De body van deze generieke eenheid wordt:

```
procedure WISSEL(EERSTE,TWEEDE : in out ELEMENT) is
  TIJDELIJK : ELEMENT;
begin
  TIJDELIJK := EERSTE;
  EERSTE    := TWEEDE;
  TWEEDE    := TIJDELIJK;
end WISSEL;
```

Op de datatypen na is de algoritme van WISSEL precies hetzelfde als die van WISSEL_INTEGER. In het generieke deel wordt ELEMENT gedeclareerd, als parameter voor de blauwdruk van het subprogramma. Overal waar eerst INTEGER stond kan nu in gedachten de parameter ELEMENT worden gesubstitueerd. We zullen nog zien dat Ada nog een andere generieke declaratiemogelijkheid heeft. Met behulp daarvan kunnen, behalve typen, ook waarden, objecten en andere subprogramma's als generieke parameters gebruikt worden.

De generieke programma-eenheid WISSEL kan nu gecompileerd worden, maar de compilatie (vertaling naar machine-instructies) zal maar ten dele worden uitgevoerd; pas als voor de generieke formele parameters, de actuele parameters worden ingevuld, vanuit een of ander aanroepend programma, pas dan kan de compilatie worden voltooid. WISSEL is maar een blauwdruk of sjabloon voor het subprogramma; het kan pas gebruikt worden als er een actuele verschijningsvorm van is gecreeërd.

Het is vrij eenvoudig een generieke eenheid te maken van een eerder ontwikkelde niet generieke eenheid. In hoofdstuk 13 beschreven we de FIFO (First In First Out) wachtrij voor INTEGER elementen. We maken hier een generieke eenheid van door voor de pakketspecificatie een generiek gedeelte te zetten, waarin een formele parameter wordt beschreven. In het verdere gedeelte van de specificatie en in de body stelt deze parameter het elementtype voor. De generieke pakketspecificatie voor FIFO kan bijvoorbeeld als volgt worden geformuleerd:

```
generic
  type ELEMENT is private;
package FIFO is
  type RIJ(LENGTE : NATURAL) is limited private;
  procedure SCHOON(BUFFER : out RIJ);
  procedure NEEM   (WAARDE : out ELEMENT; UIT : in out RIJ);
  procedure PLAATS (WAARDE : in  ELEMENT; IN  : in out RIJ);
  RIJ_VOL  : exception;
  RIJ_LEEG : exception;
private
  type LIJST is array (INTEGER range <>) of ELEMENT;
  type RIJ(LENGTE : NATURAL) is
    record
      ELEMENTEN : LIJST(0 .. LENGTE);
      VOORSTE   : INTEGER;
      ACHTERSTE : INTEGER;
    end record;
end FIFO;
```

Ook in de pakketbody moet vervolgens overal het INTEGER element worden vervangen door de formele parameter ELEMENT. We laten in de volgende paragraaf zien hoe nu deze formele programma-eenheid voor verschillende doeleinden kan worden gebruikt, door verschillende typen actuele parameters te substitueren.

Generieke verschijningsvormen

Het creëren van een actuele verschijningsvorm van een generieke programma-eenheid wordt *instantiatie* genoemd. Voor het maken van zo'n verschijningsvorm is een grootheid nodig, die een naam geeft aan de te creëren programma-eenheid. Verder moeten we actuele parameters substitueren voor de generieke parameters (net zoals actuele voor formele parameters bij het aanroepen van subprogramma's). De generieke blauwdruk wordt ingevuld via een één-eenduidige vervanging van de generieke door de actuele parameters. Vervolgens wordt het geheel verder verwerkt, alsof sprake was van een gewoon subprogramma of pakket.

De generieke eenheid WISSEL kan nu bijvoorbeeld in verschillende verschijningsvormen optreden:

```
procedure INTEGER_WISSEL   is new WISSEL(INTEGER);
procedure FLOAT_WISSEL     is new WISSEL(ELEMENT =>FLOAT);
procedure CHARACTER_WISSEL is new WISSEL(ELEMENT =>CHARACTER);
```

Dit lijkt precies op het aanroepen van subprogramma's; ook hier kunnen we de parameters benoemen, zoals in de laatste twee voorbeelden, om de leesbaarheid te vergroten. Het creëren van generieke verschijningsvormen mag overal gebeuren, waar ook subprogramma's en pakketten kunnen worden gedeclareerd. Natuurlijk moet wel de naam van de generieke eenheid zichtbaar zijn vanaf de plaats in het programma waar de verschijningsvorm wordt geschapen.

De hierboven gemaakte programma-eenheden kunnen nu gewoon gebruikt worden:

```
MIJN_LETTER,UW_LETTER : CHARACTER;
  . . .
CHARACTER_WISSEL(MIJN_LETTER,UW_LETTER);
```

14.2 Generieke Parameters



We bekijken nu een aantal categorieën generieke parameters:

- generieke typeparameters
- generieke waarde en objectparameters
- generieke subprogrammaparameters

Belangrijk: *generieke parameters zijn nooit statisch.*

Generieke typeparameters

Toen we subprogramma's en pakketten bespraken in de hoofdstukken 10 en 13, hebben we gezien dat in Ada alleen objectwaarden aan dergelijke programma-eenheden kunnen worden doorgegeven. In de vorige paragraaf illustreerden we, dat het toch wel eens gemakkelijk kan zijn als typen als parameters kunnen dienen. Naar gewone programma-eenheden is dat niet mogelijk; naar generieke eenheden echter wel.

Er zijn heel wat verschillende generieke typeparameters mogelijk, maar steeds moet de actuele parameter passen bij de formele generieke parameter. Deze eis verzekert ons van de beveiliging tegen fouten via expliciete datatypering, zelfs als de eenheden afzonderlijk worden gecompileerd. Komen de typen niet overeen, dan meldt de compiler een syntaxfout (als het tenminste mogelijk is de fout tijdens de compilatie te ontdekken), of de exceptie CONSTRAINT_ERROR treedt op (als de fout tijdens de verwerking wordt ontdekt).

In de volgende lijst zijn alle generieke parameters opgesomd, tesamen met bijpassende actuele parameters:

```
type GENERAL_PURPOSE is limited private;
  -- past bij elk datatype
type ELEMENT is private;
  -- past bij elk type waarbij waardetoekenning en test op
     (on)gelijkheid mogelijk is
type LINK is access EEN_OBJECT;
  -- past bij elk access type, verwijzend naar hetzelfde objecttype
type ENUMERATION is (<>);
  -- past bij alle discrete typen (geheeltallig en opsomming)
type INTEGER_ELEMENT is range <>;
  -- past bij elk integer type
type FIXED_ELEMENT is delta <>;
  -- past bij alle fixed-point (vaste puntnotatie) typen
type FLOAT_ELEMENT is digits <> ;
  -- past bij alle floating-point (drijvende puntnotatie) typen
type CONSTRAINED_ARRAY is array (EEN_INDEX) of EEN_ELEMENT;
  -- past bij elk begrensd array met dezelfde dimensies, dezelfde
  -- indextypen en dezelfde typecomponenten
type UNCONSTRAINED_ARRAY is array (EEN_TYPE range <>) of
                                                EEN_ELEMENT;
  -- past bij elk onbegrensd array met dezelfde dimensies, index-
     typen en componenttypen.
```

De operaties en attributen, beschikbaar voor een actueel type, kunnen in de body van een generieke eenheid ook op het overeenkomstige generieke type worden toegepast. Gebruiken we bijvoorbeeld FLOAT_ELEMENT, zoals hierboven gedeclareerd, dan kunnen binnen de body van de generieke eenheid alle voorgedefinieerde floating-point operatoren worden toegepast. Net zo zijn voor een **private** generiek type alleen de waardetoekenning en de test op (on)gelijkheid toegelaten. Voor **limited private** generieke typen zijn geen operatoren ter beschikking, behalve operatoren die werden gedefinieerd als generieke subprogrammaparameters.

Generieke typeparameters mogen in Ada, in tegenstelling tot gewone subprogrammaparameters, van elkaar afhangen. We kunnen bijvoorbeeld de volgende routine voor algemeen gebruik definiëren:

```
generic
   type ELEMENT is (<>);
   type LIJST is array(INTEGER range <>) of ELEMENT;
procedure SORT(TABEL : in out LIJST);
```

De ELEMENTen van het arraytype LIJST zijn in deze generieke subprogrammadeclaratie zelf weer een generieke formele parameter. Een verschijningsvorm van een subprogramma kan er dan zo uitzien:

```
type KLEUR is (ZWART,GROEN,ROOD,BLAUW,WIT);
type KLEUR_TABEL is array(INTEGER range <> ) of KLEUR;
package SORTEER_KLEUR is new SORT(ELEMENT => KLEUR,
                                  LIJST => KLEUR_TABEL);
```

In het volgende hoofdstuk zullen we een tamelijk ingewikkeld voorbeeld behandelen, dat gebruik maakt van generieke typeparameters. In hoofdstuk 19 zullen we nog zien, dat ook voor het verwerken van invoergegevens en het produceren van uitvoer in Ada uitgebreid van de mogelijkheden van generic gebruik wordt gemaakt.

Generieke waarde- en objectparameters

Ook waarden en objecten kunnen in Ada als generieke formele parameters worden gebruikt. De declaratie van een dergelijke parameter ziet eruit als een variabele declaratie met toevoeging van het woord in (voor een waardeparameter) of van in out (voor een objectparameter). De modus out kan niet worden gebruikt bij generieke objectparameters. Als het type van een generieke waarde- of objectparameter kan desgewenst een eerder gedeclareerde generieke typeparameter worden gebruikt.

Ook hier moet de actuele parameter weer overeenkomen met het type van de formele generieke parameter. In het geval van een waardeparameter wordt de actuele waarde in de programma-eenheid als een constante behandeld. We kunnen van deze mogelijkheid gebruik maken om bij de creatie van een verschijningsvorm van de programma-eenheid, onder- en bovengrenzen mee te geven. Op deze manier kunnen feitelijk *macro's* worden gecreëerd. (Een macro is een programmablauwdruk, die kan worden aangeroepen alsof het om een elementaire uitdrukkingsmogelijkheid in de taal ging). Het volgende pakket definieert bijvoorbeeld een generiek beeldschermstation (terminal) in de vorm van een abstract datatype:

```
generic
  REGELS            : in INTEGER := 24;
  TEKENS_PER_REGEL  : in INTEGER := 80;
package TERMINAL is ...
```

We kunnen nu een paar verschijningsvormen van dit generieke pakket maken:

```
package MICRO_TERMINAL  is new TERMINAL(24,40);
package TEKSTVERWERKER is new TERMINAL(REGELS => 66,
                              TEKENS_PER_REGEL => 132);
package PROGRAMMEURSTERMINAL is new TERMINAL;
```

Net zoals bij procedure-aanroepen kan hier van positionele parameters, benoemde parameters en default waarden gebruik worden gemaakt.

Generieke objectparameters worden via in out gedeclareerd. Aan deze parameters kan geen default waarde worden meegegeven.

Generieke subprogrammaparameters

Subprogramma's kunnen in Ada eveneens worden meegegeven als parameters voor generieke eenheden. De hierboven gedefinieerde TERMINAL kan bijvoorbeeld berichten ZENDen en ONTVANGen volgens verschillende communicatieprotocollen, afhankelijk van het type van de fysieke terminal die gebruikt wordt. Toch willen we een generiek en algemeen bruikbaar pakket definiëren, immers, vanuit het standpunt van de toepassing zijn we slechts geïnteresseerd in het beeldschermformaat. ZEND en ONTVANG kunnen nu als generieke subprogrammaparameters worden gedefinieerd:

```
generic
  REGELS             : in INTEGER := 24
  TEKENS_PER_REGEL : in INTEGER := 80;
  with procedure ZEND     (WAARDE : in  CHARACTER);
  with procedure ONTVANG(WAARDE : out CHARACTER);
package TERMINAL is . . .
```

Bij het maken van een verschijningsvorm van TERMINAL moeten we ook de namen van twee subprogramma's meegeven, die passen bij ZEND en ONTVANG. (Het actuele subprogramma moet parameters hebben van hetzelfde type, dezelfde modus en met dezelfde begrenzingen als het formele subprogramma. Als het om een functie gaat moet ook de geretourneerde waarde qua type overeenkomen.) Bijvoorbeeld:

```
procedure MICRO_ZEND     (W : out CHARACTER) is . . .
procedure MICRO_ONTVANG(W : in   CHARACTER) is . . .
package   MICRO_TERMINAL is new TERMINAL(REGELS            => 24,
                                         TEKENS_PER_REGEL => 40,
                                         ZEND      => MICRO_ZEND,
                                         ONTVANG => MICRO_ONTVANG);
```

Ook voor generieke subprogramma's kunnen default 'waarden' worden meegegeven. Dat kan in twee verschillende vormen. Ten eerste met behulp van het woorde is:

```
generic
  REGELS             : in INTEGER := 24;
  TEKENS_PER_REGEL : in INTEGER := 80;
  with procedure ZEND     (WAARDE : in  CHARACTER) is STANDAARD_ZEND;
  with procedure ONTVANG(WAARDE : out CHARACTER) is STANDAARD_ONTVANG;
package TERMINAL is . . .
```

De procedures STANDAARD_ZEND en STANDAARD_ONTVANG moeten vanaf de plaats, waar de verschijningsvorm wordt gecreëerd, zichtbaar zijn.

Ten tweede kunnen we het woordje is gebruiken, gevolgd door het samengestelde symbool '<>'. In dat geval kan de actuele parameter worden weggelaten, als tenminste een subprogramma met dezelfde naam en een passende specificatie zichtbaar is. Ga bijvoorbeeld uit van de volgende declaraties:

```
generic
  REGELS             : in INTEGER := 24;
  TEKENS_PER_REGEL   : in INTEGER := 80;
  with procedure ZEND      (WAARDE : in   CHARACTER) is <>;
  with procedure ONTVANG(WAARDE : out CHARACTER) is <>;
package TERMINAL is . . .
```

Er kan een verschijningsvorm gemaakt worden als er bijpassende ZEND en ONTVANG subprogramma's zichtbaar zijn:

```
    procedure ZEND     (W : in   CHARACTER);
    procedure ONTVANG(W : out CHARACTER);
    package EINDSTATION is new TERMINAL;
```

Het gebruikmaken van default subprogramma's in één van de twee hierboven gegeven vormen is voornamelijk gemakkelijk voor de programmeur; de programmatekst kan er wat minder duidelijk door worden en daarom is het in de meeste gevallen beter, de actuele subprogrammaparameters expliciet te benoemen.

14.3 Toepassingen Voor Ada's Generieke Programma-Eenheden



Met generieke eenheden maken we dus een blauwdruk, geraamte of sjabloon, die op het moment van compilatie nader kan worden ingevuld, al naar de behoefte van de toepassing. Ook is het met behulp van deze eenheden mogelijk, typen en subprogramma's aan andere programma-eenheden door te geven.

Een bepaalde algemeen bruikbare eenheid hoeft dus maar één keer geschreven te worden en kan dan overal worden toegepast. Als laatste voorbeeld van deze mogelijkheid komen we nog eens terug op ons pakket TRANSCENDENTE_FUNCTIES (zie hoofdstuk 13) en maken daar een generiek pakket van:

```
generic
   type RADIALEN   is digits <>;
   type RESULTAAT is digits <>;
package TRANSCENDENTE_FUNCTIES
   function COS(HOEK : in RADIALEN) return RESULTAAT;
   function SIN (HOEK : in RADIALEN) return RESULTAAT;
   function TAN(HOEK : in RADIALEN) return PESULTAAT;
end TRANSCENDENTE_FUNCTIES;
```

Van dit pakket kunnen we verschijningsvormen creëren met functies tot elke gewenste nauwkeurigheid.

Oefeningen

1. Formuleer een verschijningsvorm voor de generieke eenheid TRANSCENDENTE_FUNCTIES, waarin RADIALEN en RESULTAAT een precisie van 10 significante cijfers hebben. Gebruik eerst de positionele parameternotatie en dan benoemde parameters.

*2. Welke wijzigingen moeten er in de body van TRANSCENDENTE-_FUNCTIES worden aangebracht, om aan de specificaties van opgave 1 te kunnen voldoen?

3. Schrijf de specificatie voor een matrixpakket, waarmee matrices kunnen worden opgeteld, afgetrokken en vermenigvuldigd. Voeg vervolgens een generiek gedeelte toe, zodat het type van de matrixelementen als parameter kan worden meegegeven.

4. Wijzig de specificatie van de in het vorige hoofdstuk geformuleerde LEXICAL_ANALYZER, door een generiek gedeelte toe te voegen, zodat de gebruiker kan aangeven welke tekens uit de verzameling ALPHA en de verzameling DIGITS geaccepteerd mogen worden. Maak een verschijningsvorm om het gebruik van dit pakket te illustreren.

15 HET DERDE ONTWERPPROBLEEM: HET GENERIEKE PAKKET SET

Pakketten en generieke programma-eenheden kunnen ons helpen bij het beheersbaar houden van complexe programmatuur. Het wordt nu mogelijk algemeen bruikbare softwaremodulen te ontwikkelen en daadwerkelijk een software-industrie op te bouwen. Binnen toepassingen kan van deze modulen gebruik worden gemaakt op dezelfde manier als van elementaire taalelementen.

In dit hoofdstuk zullen we een dergelijke algemeen bruikbare module ontwikkelen. In Ada zijn geen verzamelingen en operaties op verzamelingen gedefinieerd, zoals bijvoorbeeld in Pascal; we gaan daarom een pakket voor het manipuleren van verzamelingen ('sets') maken. Objecten kunnen binnen verschillende verzamelingen van verschillend type zijn, dus ligt het voor de hand van ons verzamelingpakket een generieke eenheid te maken. Nu we de vorm van pakketten en generieke eenheden in de vorige hoofdstukken hebben bestudeerd, kunnen we dit pakket volledig uitwerken. In hoofdstuk 18 zullen we het pakket binnen een ander ontwerpprobleem nog eens gebruiken.

15.1 Definieer Het Probleem



In wiskundige zin is een *verzameling* een collectie van objecten, *elementen* geheten. We kunnen die objecten desgewenst namen geven. Bijvoorbeeld

[koperinstrumenten,slagwerk,strijkinstrumenten,houtblazers]

Hiermee definieerden we een *universum* voor alle volgende verzamelingen; alleen de vier objecten uit het universum kunnen daarin voorkomen. Als in een verzameling niet alle vier elementen uit het universum voorkomen, dan heet die verzameling een deelverzameling (*subset*). Een verzameling zonder elementen heet *leeg* en wordt de nulverzameling genoemd, ook wel weergegeven door:

{ }

Er zijn een aantal operaties op verzamelingen mogelijk. Gegeven de verzamelingen A, B en C, kunnen we waarden aan die verzamelingen toekennen:

A ← {koperinstrumenten,houtblazers}

B ← { }

C ← A (dus C = {koperinstrumenten,houtblazers})

We kunnen elementen aan een verzameling toevoegen en zelfs twee verzamelingen bij elkaar optellen. Deze laatste operatie heet de *vereniging* van twee verzamelingen en kan bijvoorbeeld door de '+' operator worden aangegeven:

A ← A + {strijkinstrumenten} (A = {koperinstrumenten, strijkinstrumenten,houtblazers})

B ← {koperinstrumenten} + B (B = {koperinstrumenten})

C ← A + B (C = {koperinstrumenten,strijkinstrumenten,houtblazers})

Het laatste voorbeeld laat zien wat er gebeurt als de vereniging wordt bepaald van twee verzamelingen, waarin gelijke elementen voorkomen. A en B bevatten beide het element 'koperinstrumenten', toch komt in C dit element maar één keer voor. Een verzameling bepaalt alleen maar het wel of niet voorkomen van een element.

We kunnen een element van een verzameling aftrekken, of de ene verzameling van de andere aftrekken. Deze operatie heet het *verschil* tussen twee verzamelingen en kan door het teken '-' worden weergegeven:

C ← A - {strijkinstrumenten} (C ={koperinstrumenten,houtblazers})

B ← A - B (B ={strijkinstrumenten,houtblazers})

A ← C - {slagwerk} (A ={koperinstrumenten,houtblazers})

In het laatste voorbeeld probeerden we de verzameling C te verminderen met een element dat niet in C aanwezig was. Dit is toegelaten: het verwijderen van een object dat niet voorkomt heeft geen effect.

Soms is het nodig vast te stellen, welke objecten twee verzamelingen gemeenschappelijk hebben. Deze operatie heet het bepalen van de *doorsnede* van die verzamelingen en dit kan door het teken '*' worden weergegeven. Ga bijvoorbeeld uit van de waarden van A en B uit het vorige voorbeeld en bepaal:

C ← A * B (C = {houtblazers})

'houtblazers' was het enige gemeenschappelijke element van A en B.

Ook wil men verzamelingen op onderlinge gelijkheid en op de vraag of de ene verzameling een deelverzameling van de andere is,

kunnen testen. Er wordt een onderscheid gemaakt tussen *deelverzamelingen* en *echte deelverzamelingen*. Een verzameling is een deelverzameling van een andere verzameling als alle elementen van de eerste verzameling ook elementen van de tweede zijn. Er is sprake van een echte deelverzameling als de tweede verzameling elementen bevat, die niet in de eerste voorkomen. Een voorbeeld:

A ← {slagwerk,houtblazers}

B ← A

C ← {slagwerk}

B en C zijn beide deelverzamelingen van A, maar alleen C is een echte deelverzameling van A. We zullen het teken '<' gebruiken voor 'is echte deelverzameling van' en het samengestelde teken '<=' voor 'is deelverzameling van '. Nu geldt:

B <= A (deze bewering is TRUE) en C < A (dit is eveneens TRUE)

Elke verzameling is een deelverzameling, maar geen echte deelverzameling, van zichzelf.

Tenslotte moeten we nog kunnen testen of een element tot een bepaalde verzameling hoort en willen we kunnen bepalen hoeveel elementen een bepaalde verzameling heeft. Dit aantal wordt het *kardinaalgetal* van de verzameling genoemd. Verder willen we nog kunnen beslissen of een bepaalde verzameling leeg is. Als we uitgaan van de volgende verzamelingen:

A ← {slagwerk,strijkinstrumenten, houtblazers}

B ← {koperinstrumenten,strijkinstrumenten, houtblazers}

C ← { }

dan is 'koperinstrumenten' geen element van A, is het kardinaalgetal van B gelijk aan drie en is de uitkomst van de test of C leeg is gelijk aan TRUE.

Na deze korte inleiding in de verzamelingenleer, gaan we een pakket ontwerpen, met behulp waarvan de hier genoemde operaties op willekeurige verzamelingen kunnen worden toegepast.

15.2 Ontwikkel Een Informele Strategie



In dit stadium van de probleemoplossing verdiepen we ons nog niet in de vraag hoe verzamelingen opgebouwd kunnen worden met behulp van Ada's elementaire typen. We benaderen het verzamelingbegrip eerst vanuit het standpunt van de gebruiker. Pas later wordt

de implementatie gekozen, en dit heeft het voordeel, dat deze kan worden gewijzigd zonder dat de gebruiker hier iets van hoeft te merken.

De probleemspecificaties kunnen als volgt worden geformuleerd:

> Het begrip verzameling moet kunnen worden gedefinieerd en we moeten het universum van de verzameling kunnen definiëren. Verder dienen de op verzamelingen gebruikelijke operaties beschikbaar te zijn. En wel: vereniging en verschil (van een verzameling met een andere verzameling of met een element) en doorsnede (van twee verzamelingen). Verder: tests op gelijkheid en ongelijkheid, op deelverzameling en echte deelverzameling, bepaling van kardinaalgetal, voorkomen van een element en test op leeg zijn van een verzameling. Verder moet waardetoekenning aan verzamelingen mogelijk zijn en moet er een constante, die de lege verzameling voorstelt, beschikbaar zijn.

15.3 Formaliseer De Strategie



We gaan de informele strategie nu formeel in Ada formuleren:

Bepaal de objecten en hun attributen

We identificeren nu de voor ons probleem relevante objecten:

> Het begrip verzameling moet kunnen worden gedefinieerd en we moeten het universum van de verzameling kunnen definiëren. Verder dienen de op verzamelingen gebruikelijke operaties beschikbaar te zijn. En wel: vereniging en verschil (van een verzameling met een andere verzameling of met een element) en doorsnede (van twee verzamelingen). Verder: tests op gelijkheid en ongelijkheid, op deelverzameling en echte deelverzameling, bepaling van kardinaalgetal, voorkomen van een element en test op leeg zijn van een verzameling. Verder moet waardetoekenning aan verzamelingen mogelijk zijn en moet er een constante, die de lege verzameling voorstelt, beschikbaar zijn.

In dit stadium van de formulering blijken de volgende objecten voor te komen:

- SET (= verzameling)
- NULL_SET (= lege verzameling)
- UNIVERSUM

(We gebruiken kortheidshalve de Engelse term SET, die bijvoorbeeld ook in de programmeertaal Pascal voorkomt.)

Omdat we verschillende verzamelingen met verschillende typen elementen willen kunnen definiëren, zullen we een abstract type SET

creëren. Vervolgens kunnen we dan objecten van dit type definiëren. Het UNIVERSUM van ons pakket wordt gevuld op het moment van de creatie van een actuele verschijningsvorm.

Bepaal de operaties op de objecten

We onderstrepen nu de operaties (werkwoorden) in onze probleemformulering:

> **Het begrip verzameling moet kunnen worden gedefinieerd en we moeten het universum van de verzameling kunnen definiëren. Verder dienen de op verzamelingen gebruikelijke operaties beschikbaar te zijn. En wel: vereniging en verschil (van een verzameling met een andere verzameling of met een element) en doorsnede (van twee verzamelingen). Verder: tests op gelijkheid en ongelijkheid, op deelverzameling en echte deelverzameling, bepaling van kardinaalgetal, voorkomen van een element en test op leeg zijn van een verzameling. Verder moet waardetoekenning aan verzamelingen mogelijk zijn en moet er een constante, die de lege verzameling voorstelt, beschikbaar zijn.**

We kunnen nu de operaties opsommen:

- SET
  - = (gelijkheid)
  - /= (ongelijkheid)
  - * (doorsnede)
  - + (vereniging van twee verzamelingen of van een verzameling met een element)
  - - (verschil van twee verzamelingen of van een verzameling met een element)
  - < (echte deelverzameling)
  - <= (deelverzameling)
  - IS_ELEMENT
  - IS_LEEG
  - AANTAL_IN
- NULL_SET
- UNIVERSUM

Er werden geen speciale operaties voor de lege verzameling (NULL_SET) gedefinieerd: de lege verzameling is een object van het type SET als ieder ander. Ook op UNIVERSUM zijn geen operaties gedefinieerd; dit gebeurt bij de creatie van de SET objecten.

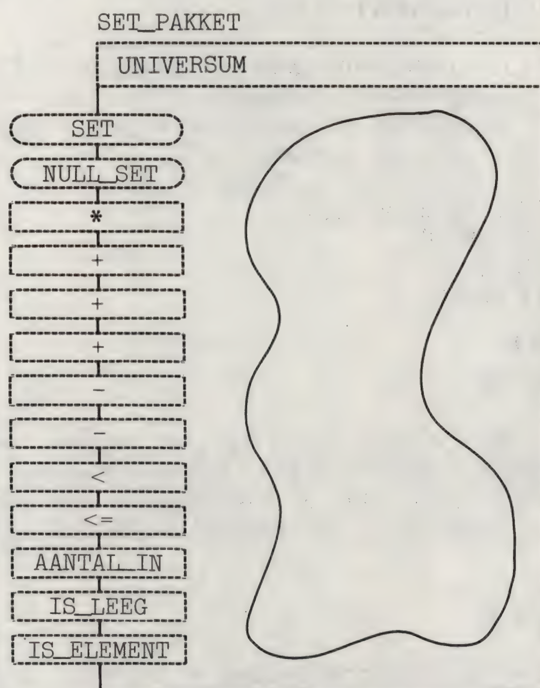
Bepaal de interfaces

Nu we de objecten en de operaties daarop kennen, is de volgende stap het vastleggen van de wijze van communicatie of de interface met de gebruiker. Omdat we een abstract datatype SET definiëren,

zullen in ieder geval de typenaam en de bijbehorende operaties uit het pakket moeten worden geëxporteerd. Met behulp van de typenaam kan de gebruiker nu objecten van het type SET creëren. Verder moet de constante NULL_SET, de lege verzameling, gebruikt kunnen worden en dus uit het pakket worden geëxporteerd. Hoe de operaties precies functioneren blijft voor de gebruiker van het pakket verborgen en dizelfde geldt voor de implementatie van de verzameling zelf. In figuur 15-1 is het generieke pakket SET nog eens grafisch weergegeven.

We willen het pakket voor verzamelingen van alle mogelijke elementen kunnen gebruiken en maakten er daarom een generiek pakket van, met UNIVERSUM als generieke parameter. De operaties voor waardetoekenning en test op gelijkheid en ongelijkheid behoeven niet expliciet te worden geëxporteerd: we zullen SET als type 'private' declareren en deze operaties zijn dan impliciet toegelaten. Uit de figuur blijkt dat er maar liefst drie '+' operatoren en twee '-' operatoren worden geëxporteerd. Dit is wel degelijk nodig, want de volgende operaties moeten mogelijk zijn:

- tussen verzameling en element '+' '-'
- tussen element en verzameling '+'
- tussen twee verzamelingen '+' '-' '*'



Figuur 15-1 Ontwerp voor SET_PAKKET

Dezelfde symbolen worden voor in bepaalde opzichten verschillende operaties gebruikt; de operatoren worden 'overladen'. Dit biedt in Ada geen problemen; de regels in de taal voor het bepalen van de bedoelde operaties kunnen steeds zorgen dat het juiste subprogramma wordt aangeroepen.

We gaan nu de code uitschrijven voor de zichtbare gedeelten van SET_PAKKET:

```
generic
  type UNIVERSUM is (<>);
package SET_PAKKET is
  --
  type SET is private;
  NULL_SET : constant SET;
  --
  function "*"  (SET_1   : in SET      ; SET_2   : in SET)       return SET;
  function "+"  (ELEMENT : in UNIVERSUM; SET_1   : in SET)       return SET;
  function "+"  (SET_1   : in SET      ; SET_2   : in SET)       return SET;
  function "+"  (SET_1   : in SET      ; ELEMENT : in UNIVERSUM) return SET;
  function "-"  (SET_1   : in SET      ; SET_2   : in SET)       return SET;
  function "-"  (SET_1   : in SET      ; ELEMENT : in UNIVERSUM) return SET;
  function "<"  (SET_1   : in SET      ; SET_2   : in SET)       return BOOLEAN;
  function "<=" (SET_1   : in SET      ; SET_2   : in SET)       return BOOLEAN;
  --
  function IS_ELEMENT(ELEMENT : in UNIVERSUM; VAN_SET : in SET) return BOOLEAN;
  function IS_LEEG    (SET_1 : in SET)                          return BOOLEAN;
  subtype AANTAL is INTEGER range 0 .. (UNIVERSUM'POS(UNIVERSUM'LAST) -
                                        UNIVERSUM'POS(UNIVERSUM'FIRST) + 1);
  function AANTAL_IN(SET_1 : in SET) return AANTAL;
private
  type SET is . . .
  NULL_SET : . . .
end SET_PAKKET;
```

De generieke parameter UNIVERSUM gaven wij aan met het symbool '<>'. Dit kan, omdat de actuele parameter altijd een discreet type zal zijn (een integer of een opsomming). De NULL_SET definieerden we als een constante; de waarde ervan is echter niet gedefinieerd. Omdat NULL_SET een verschijningsvorm is van een private type is die waarde buiten het pakket ook niet zichtbaar.Men noemt dit een *uitgestelde constante*: de waarde zal worden toegekend bij de uitwerking van het private gedeelte van het pakket.

De subprogrammaspecificaties voor de overladen operatoren vertonen opvallend veel onderlinge overeenkomst. In dit geval heeft de taal wel erg veel omhaal van woorden nodig, maar het is nu eenmaal nodig alle operaties expliciet te definiëren, zelfs als alleen de volgorde van parameters maar verschillend is.

Het subprogramma AANTAL_IN moet een geheeltallige grootheid retourneren, die begrensd wordt door het aantal mogelijke waarden binnen het UNIVERSUM. De ondergrens is in ieder geval 0, want de verzameling kan de NULL_SET zijn. De bovengrens bepalen we door eerst het attribuut UNIVERSUM'LAST te bepalen. Dit geeft de waarde van het laatste element van het UNIVERSUM. Als hier nu UNIVERSUM'POS op toe wordt gepast, krijgen we de positie van dat

element (dat wil zeggen het volgnummer). Vervolgens doen we hetzelfde voor het eerste element en trekken deze waarde van de eerste af. Dit verschil, vermeerderd met één is juist het totale aantal elementen in het UNIVERSUM.

We hebben nu een algemeen toepasbaar pakket voor ieder type verzameling. We geven een voorbeeld. Stel, het universum is:

```
type KLEUR is (ROOD,GROEN,BLAUW,WIT);
```

We kunnen dan bepalen:

```
KLEUR'LAST                                        => WIT
KLEUR'POS(KLEUR'LAST)                             => 3
KLEUR'FIRST                                       => ROOD
KLEUR'POS(KLEUR'FIRST)                            => 0
KLEUR'POS(KLEUR'LAST)-KLEUR'POS(KLEUR'FIRST)+1    => 4
```

en dit is precies het antwoord dat we zochten.

We hebben nu de zichtbare gedeelten van het pakket ontworpen, maar de Ada syntaxis vereist, dat ook de private eenheden op dit niveau worden uitgewerkt. Weliswaar kan de gebruiker met deze implementatie niets doen, maar de programmatekst is natuurlijk wel leesbaar. Volledigheidshalve completeren we daarom de pakketspecificatie, door SET als een array van BOOLEAN elementen te implementeren, met als totale lengte het aantal elementen in UNIVERSUM. We krijgen dan:

```
private
  type SET is array(UNIVERSUM) of BOOLEAN;
  NULL_SET : constant SET := SET'(others => FALSE);
```

De NULL_SET of lege verzameling is dus een object van het type SET zonder elementen; de aanwezigheid van elk element is FALSE.

Overal waar dit generieke pakket zichtbaar is binnen een programma, kan door de gebruiker een verschijningsvorm worden gecreëerd voor het gewenste verzamelingstype. Gaan we bijvoorbeeld uit van de volgende typen:

```
type INSTRUMENTEN  is (KOPERINSTRUMENTEN,SLAGWERK,
                       STRIJKINSTRUMENTEN,HOUTBLAZERS);
type FRUIT         is (APPEL,BANAAN,PEER,SINAASAPPEL);
type HEX_TEKEN     is ('A','B','C','D','E','F');
type CIJFER        is range 0 .. 9;
```

dan kunnen de volgende pakketten worden benoemd:

```
package ORKEST_SET  is new SET_PAKKET(UNIVERSUM => INSTRUMENTEN);
package VRUCHTEN    is new SET_PAKKET(FRUIT);
package HEX_SET     is new SET_PAKKET(UNIVERSUM => HEX_TEKEN);
package CIJFER_SET  is new SET_PAKKET(CIJFER);
```

We gebruikten zowel de benoemde als de positionele parameternotatie. De eerste notatie wordt aanbevolen vanwege de betere leesbaarheid.

Nu kan van de mogelijkheden uit SET_PAKKET gebruik worden gemaakt:

```
declare
  use ORKEST_SET;
  INSTRUMENTARIUM : ORKEST_SET.SET := NULL_SET;
begin
  . . .
  INSTRUMENTARIUM := HOUTBLAZERS + INSTRUMENTARIUM;

end;
```

In feite hadden we wegens de use clausule, bij de declaratie van INSTRUMENTARIUM, niet nog eens de kwalificatie ORKEST_SET hoeven te gebruiken. Vaak wordt dit, zoals we hier deden, toch gedaan, ook weer ter vergroting van de begrijpelijkheid. Ook is hier te zien, dat de verzameling expliciet door de gebruiker moet worden geïnitialiseerd. In hoofdstuk 18 zullen we een toepassing van dit pakket behandelen.

Programmeer de operaties

We hebben nu de wijze van communicatie, de 'interface' van SET-_PAKKET met de buitenwereld beschreven en nu gaan we de operaties, die door het pakket naar deze buitenwereld worden geëxporteerd, verder uitwerken. We kozen voor een array, om de elementen van de verzameling voor te stellen en daarom kunnen we nu handig gebruik maken van de operaties die voor BOOLEAN arrays in Ada zijn gedefinieerd, zoals de and, or en not operatie.

De SET_PAKKET body heeft de volgende vorm:

```
package body SET_PAKKET is
  --
  -- bodies van de subprogramma's die in de pakketspecificatie
     werden gedeclareerd
  --
end SET_PAKKET;
```

Er hoeven in dit geval geen beginwaarden aan grootheden te worden toegekend en daarom hoeft de body niet met een rij instructies te beginnen.

We gaan nu eerst de doorsnede operator '*' uitwerken. Als deze operatie wordt uitgevoerd, dan is het resultaat een verzameling die de elementen uit beide voor de operatie gebruikte verzamelingen bevat. Hier kan de logische and operatie worden toegepast, die gedefinieerd is voor BOOLEAN arrays:

```
function "*" (SET_1 in SET; SET_2 in SET) return SET is
-- doorsnede van twee verzamelingen
begin
  return (SET_1 and SET_2);
end "*";
```

De "*" operator is nu overladen met een operatie op verzamelingen. (Alle operatorsymbolen, behalve "/=" (niet gelijk aan) kunnen worden overladen. De operator "/=" wordt automatisch overladen, als de "=" operator wordt overladen. Het waardetoekenningsteken ":=" kan nooit overladen worden en heeft dus nooit meer dan één betekenis.)

De verenigingsoperator "+" kan nu op soortgelijke wijze worden uitgewerkt:

```
function "+" (ELEMENT : in UNIVERSUM; SET_1 : is SET) return SET
-- vereniging
  WAARDE_VERZAMELING : SET := SET_1;
begin
  WAARDE_VERZAMELING(ELEMENT) := TRUE;
  return WAARDE_VERZAMELING;
end "+";
--
function "+" (SET_1 : in SET; SET_2 : in SET) return SET is
-- vereniging
begin
  return (SET_1 or SET_2);
end "+";
--
function "+" (SET_1 : in SET; ELEMENT : in UNIVERSUM) return SET
--vereniging
  WAARDE_VERZAMELING : SET := SET_1;
begin
  WAARDE_VERZAMELING(ELEMENT) := TRUE;
  return WAARDE_VERZAMELING;
end "+";
```

Formele parameters bij functies zijn altijd van het in type en er kan dus geen nieuwe waarde aan worden toegekend. Vandaar dat in twee gevallen in het hierboven gegeven voorbeeld van een lokale hulpvariabele WAARDE_VERZAMELING gebruik moest worden gemaakt om het resultaat te retourneren. Opvallend is ook dat de derde "+" operator op de volgorde van de parameters na, identiek is aan de eerste. De derde operator had dus kunnen worden geïmplementeerd met een aanroep van de eerste, na verwisseling van de operanden, maar dit was voor de leesbaarheid en de onderhoudbaarheid weer niet bevorderlijk geweest.

De verschiloperator "-", waarmee een element uit een verzameling kan worden verwijderd, kan als volgt worden geformuleerd:

```
function "-" (SET_1 : in SET; ELEMENT : in UNIVERSUM) return SET is
-- verschiloperator
  WAARDE_VERZAMELING : SET := SET_1;
begin
  WAARDE_VERZAMELING(ELEMENT) := FALSE;
  return WAARDE_VERZAMELING;
end "-";
--
function "-" (SET_1 : in SET; SET_2 : in SET) return SET is
-- verschiloperator
begin
  return SET_1 and (not SET_2));
end "-";
```

Het verschil tussen twee verzamelingen komt overeen met de doorsnede van de eerste verzameling met het complement van de tweede. We geven een voorbeeld:

SET_1 ← {houtblazers, slagwerk}
SET_2 ← {houtblazers, strijkinstrumenten}

De zogenaamde waarheidstabel voor het verschil van SET_1 en SET_2, waarin voor elk element wordt aangegeven of dit wel (TRUE) of niet (FALSE) in de verschilverzameling voorkomt, kan als volgt worden opgebouwd:

ELEMENT	SET_1	SET_2	not SET_2	SET_1 and (not SET_2)
houtblazers	TRUE	TRUE	FALSE	FALSE
slagwerk	TRUE	FALSE	TRUE	TRUE
koperinstrumenten	FALSE	FALSE	TRUE	FALSE
strijkinstrumenten	FALSE	TRUE	FALSE	FALSE

Het verschil komt dus juist overeen met een verzameling, die alleen het ene element *slagwerk* bevat.

De deelverzamelingsrelaties kunnen op ongeveer dezelfde manier worden geprogrammeerd. We maakten een verschil tussen een 'echte' deelverzameling, "<" en een deelverzameling "<=". Het bepalen van de deelverzamelingsrelatie kan gebeuren met behulp van de doorsnede-operatie. Als de doorsnede van twee verzamelingen gelijk is aan de eerste verzameling, dan is de eerste verzameling een deelverzameling van de tweede. Als verder deze doorsnede niet gelijk is aan de tweede verzameling, dan is er sprake van een 'echte' deelverzameling. We krijgen dan:

```
function "<=" (SET_1 : in SET; SET_2 : in SET) return BOOLEAN is
-- deelverzamelingsoperator
  WAARDE_VERZAMELING : SET := (SET_1 and SET_2);
begin
  return (WAARDE_VERZAMELING = SET_1);
end "<=";
--
function "<" (SET_1 : in SET; SET_2 : in SET) return BOOLEAN is
-- echte deelverzamelingaoperator
  WAARDE_VERZAMELING : SET := SET (SET_1 and SET_2);
begin
  return ((WAARDE_VERZAMELING = SET_1) and (WAARDE_VERZAMELING /= SET_2));
end "<";
```

De overige operatoren kunnen even makkelijk worden uitgewerkt. IS_ELEMENT wordt bepaald door het al of niet voorkomen van een element in de verzameling. IS_LEEG is eenvoudig een test op gelijkheid met de NULL_SET. AANTAL_IN kan worden berekend door het aantal elementen in de verzameling TRUE te tellen. We werken dit als volgt uit:

```
function IS_ELEMENT(ELEMENT : in UNIVERSUM; VAN_SET : in SET)
          return BOOLEAN is
-- test op voorkomen element in verzameling
begin
  return VAN_SET(ELEMENT);
end IS_ELEMENT;
--
function IS_LEEG(SET_1 : in SET) return BOOLEAN is
-- test op leeg zijn verzameling
begin
  return (SET_1 = NULL_SET);
end IS_LEEG;
--
function AANTAL_IN(SET_1 in SET) return AANTAL is
-- bepalen kardinaalgetal
  TEL : NATURAL := 0;
begin
  for INDEX in UNIVERSUM
    loop
      if SET_1(INDEX) then
        TEL := TEL + 1;
      end if;
    end loop;
  return TEL;
end AANTAL_IN;
```

Hiermee is ons SET_PAKKET klaar. De code staat nogal verspreid door dit hoofdstuk en daarom is het geheel nog eens in Appendix F opgenomen.

We hebben nu Ada als sequentiële taal (dat wil zeggen als taal, waarin instructies worden gespecificeerd, die achtereenvolgens

moeten worden uitgevoerd) grotendeels behandeld. De STEELMAN specificaties spreken echter al van een taal, geschikt voor real-time in grotere systemen ingebedde computersystemen. In dergelijke systemen is vaak sprake van gelijktijdig of parallel optredende verschijnselen en bewerkingen. In het volgende pakket van dit boek zullen we Ada's mogelijkheden voor het specificeren en besturen van parallelle processen aan een nauwgezette analyse onderwerpen.

Oefeningen

1. Ga na dat de AANTAL_IN operator correct werkt voor een verzameling bestaande uit een interval van geheeltallige grootheden.

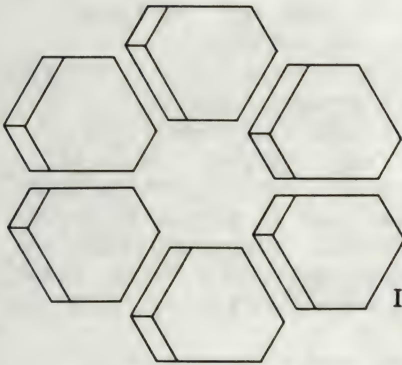
2. Maak een type SOORT_COMPUTER en creëer een verschijningsvorm van SET_PAKKET voor dit type. Declareer vier objecten van dit type verzamelingen.

3. Schrijf een subprogramma dat als invoerparameter een verzamelingsobject accepteert en dat de waarde TRUE retourneert als deze verzameling een even aantal elementen bevat, terwijl in alle andere gevallen de waarde FALSE wordt geretourneerd.

4. Is het mogelijk een verzameling van de reële getallen op te bouwen met behulp van ons SET_PAKKET? Licht uw antwoord toe.

Pakket 6

PARALLELLE REAL-TIME VERWERKING



If it were done, when 'tis done,
then 'twere well
It were done quickly

Shakespeare
Macbeth [1]

16 TAKEN

In werkelijkheid verlopen vaak een aantal processen gelijktijdig of parallel. De automatische piloot in een vliegtuig bijvoorbeeld, houdt voortdurend oog op een aantal grootheden, zoals snelheid en hoogte, is tegelijkertijd attent op eventuele nieuwe opdrachten van de bemanning en regelt daarbij nog motortoerental en hoogteroeren. De besturing van deze processen kan verder op willekeurige momenten worden onderbroken voor het uitvoeren van andere taken, zoals navigatieberekeningen.

De meeste hogere programmeertalen zijn niet erg geschikt voor het formuleren van de besturing van dergelijke parallelle activiteiten. Meestal is men genoodzaakt direct van in het operating system opgenomen faciliteiten gebruik te maken, of men moet zelf een aantal routines schrijven ter synchronisatie van een aantal taken.

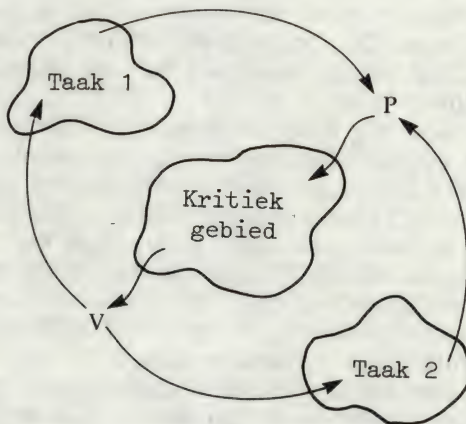
Geen van beide oplossingen is eigenlijk aanvaardbaar. Buiten de hogere programmeertaal om gaan, maakt overdraagbaarheid van de programmatuur naar andere computersystemen zeer moeilijk en ditzelfde geldt voor de onderhoudbaarheid. Het schrijven van deze zogenaamde multitasking software is trouwens evenmin een eenvoudige bezigheid. Dit laatste geldt zeker wanneer in assembler geprogrammeerd moet worden; het is dan moeilijk, zo niet onmogelijk, om tijdsafhankelijke relaties aan te geven en communicatie tussen verschillende taken op een betrouwbare manier te bestuderen. (Zie ook nog eens hoofdstuk 4.)

In Ada is een *taak* (Engels: task) een programma-eenheid, die gelijktijdig actief is met andere eenheden. Op logisch niveau kan een oplossing van een probleem vaak worden voorgesteld met behulp van een aantal verschillende taken, die tegelijkertijd moeten worden uitgevoerd. De fysieke oplossing kan bestaan uit verwerking van deze taken door een multiprocessorsysteem, of uit quasi parallelle verwerking door één processor. Hoe de uiteindelijke fysieke verwerking ook plaatsvindt, het is vaak een voor de hand liggende en logische gedachte, uit te gaan van een aantal parallel uit te voeren taken. Dit parallellisme willen we daarom ook in de formulering van onze probleemoplossing tot uitdrukking kunnen brengen. Net zoals we het vanzelfsprekend vinden met reële getallen te werken en niet met rijtjes bits, net zo gewoon moet het worden om met parallelle taken te werken, in plaats van met zuiver sequentiële oplossingen.

Meestal zijn taken niet volledig onderling onafhankelijk; het moet dus mogelijk zijn taken onderling te laten communiceren. De wijze van communicatie moet helder zijn en volkomen betrouwbaar. Bij wijze van voorbeeld kiezen we een systeem dat twee taken heeft. De eerste taak bewaakt een toetsenbord en verzamelt ingetoetste tekstregels. (Deze taak noemen we de PRODUCENT.) De tweede taak verzendt de in een buffer opgeslagen tekens via een modem naar elders. (Dit is de CONSUMENT.) Beide taken verlopen asynchroon. Van het toetsenbord kan minuten lang geen enkel bericht komen en dan kunnen vervolgens de gegevens plotseling met 80 woorden per minuut binnenstromen. Omdat van een modem gebruik wordt gemaakt (bijvoorbeeld voor verzending langs een telefoonlijn), moet ook de communicatie met het modem worden geregeld en kan het zelfs nodig zijn een bericht opnieuw te verzenden indien correcte ontvangst niet bevestigd wordt.

Communicatie tussen taken kan op twee verschillende manieren worden georganiseerd. De eerste methode lijkt op het sturen van een brief via de post. De PRODUCENT spaart hoeveelheden toetsaanslagen op en plaatst deze, als een hele regel is ingevoerd, in een voor beide taken gemeenschappelijk gebied: de *brievenbus*. Er wordt nu door de PRODUCENT een vlag gezet, om aan te geven dat er een bericht in de brievenbus zit. De CONSUMENT haalt deze berichten één voor één uit de brievenbus.

Deze communicatie wordt zo geregeld, dat het gemeenschappelijk gebied maar door één taak op ieder tijdstip betreden kan worden; er is sprake van *wederzijdse uitsluiting*. Als beide tegelijk dit gebied willen binnengaan, dan moet één van beide taken wachten en de andere voor laten gaan. De CONSUMENT wacht bij de brievenbus zolang deze leeg is. De PRODUCENT plaatst een bericht in de bus en begint vervolgens meteen aan de produktie van het volgende bericht.



Figuur 16-1 Communicatie tussen taken geregeld door een semafoor.



Figuur 16-2 Taken als communicerende sequentiële processen.

Een dergelijk communicatiesysteem wordt binnen programma's opgebouwd met behulp van een *monitor* of met een *semafoor*. In figuur 16-1 is te zien hoe de brievenbus beschouwd kan worden als een kritiek gebied, dat op ieder moment maar door één taak kan worden binnengegaan en waarbinnen zich maar één taak mag bevinden. De vlag is hier geïmplementeerd als een semafoor (een soort seinpaal of seinlamp), die door de operatie V op 'Vrij' gezet wordt en door de operatie P op 'Bezet' wordt gezet. In eenvoudige gevallen geeft communicatie met behulp van semaforen weinig problemen, maar in complexere situaties levert de implementatie nogal wat moeilijkheden op.

Als er in plaats van van één PRODUCENT en één CONSUMENT sprake is van meerdere van dergelijke processen, dan moeten bepaalde prioriteitsregels worden opgesteld voor het toewijzen van processortijd en van randapparatuur. Communicatie tussen taken via semaforen verloopt asynchroon en het is daarom moeilijk te voldoen aan bepaalde randvoorwaarden, zoals: 'de via het toetsenbord ingevoerde gegevens worden via het modem verstuurd binnen 500 milliseconden, nadat een complete regel is ontvangen'. De ene taak is niet op de hoogte van het bestaan van de andere taken (elke taak 'ziet' alleen de semafoor, niet de andere taken) en het is daarom ook moeilijk om de ene taak te laten signaleren, dat er iets fout is gegaan binnen een andere taak. Zo zou de CONSUMENT vast kunnen lopen, terwijl de PRODUCENT rustig doorgaat met de produktie van regels tekst, die dan verloren gaan.

Een andere benadering voor de interactie van taken, is die van de communicerende sequentiële processen [1], zoals die in figuur 16-2 is weergegeven. De communicatie verloopt nu synchroon, ongeveer zoals een gesprek tussen twee personen. Als de PRODUCENT een regel tekst heeft voortgebracht, richt deze een verzoek tot communicatie aan de CONSUMENT. Zodra de CONSUMENT dit verzoek accepteert, wordt het bericht verzonden en hervatten beide taken hun eigen werkzaamheden, totdat de PRODUCENT opnieuw aangeeft voor communicatie gereed te zijn.

Een dergelijke expliciete synchronisatie tussen communicerende processen wordt een *rendez-vous* genoemd. Als een taak bereid is tot zenden of ontvangen voordat zijn partner bij het ontmoetingspunt is aangeland, dan kan de eerste ofwel voor onbepaalde tijd op de tweede wachten, ofwel gedurende een vaste periode wachten,

ofwel omzien naar een andere taak die wel tot communicatie bereid is. Een van de voordelen van deze methode is dat hij betrouwbaarder is dan het werken met semaforen. Als een bepaalde taak vastloopt of niet meer reageert, dan kan dit door een andere taak ontdekt worden en kan dienovereenkomstig actie worden ondernomen. Omdat hier de communicatie ook als synchronisatiemiddel wordt gebruikt, kunnen ook tijdscriteria gemakkelijker gecontroleerd worden.

Ada werd ontworpen voor in anderssoortige systemen ingebedde computersystemen en de taal bevat daarom mogelijkheden om de besturing van meerdere parallel lopende taken te formuleren en uit te voeren via de methode van de communicerende sequentiële processen. In Ada is het daarbij niet nodig, zoals wel het geval is in veel andere hogere talen, om buiten de taal om te gaan. In dit hoofdstuk zullen we laten zien hoe een en ander in Ada wordt aangepakt.

16.1 Taken In Ada: Vorm

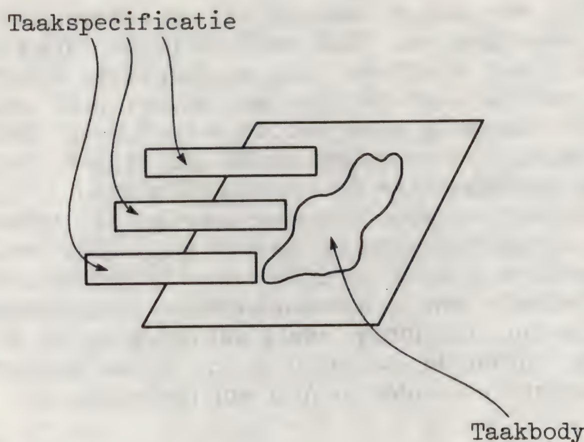


De taak is één van de drie basis programma-eenheden in Ada. De andere twee zijn het subprogramma en het pakket. Het hoofdprogramma wordt impliciet als taak beschouwd, maar daarbinnen en binnen lagere programma-eenheden kunnen expliciet taken worden gedeclareerd. Taken kunnen worden ingevoerd binnen het declaratiegedeelte van elke eenheid, zoals een blok, een subprogrammabody, een taakbody, of binnen een bibliotheekpakket. Een taak kan niet afzonderlijk worden gecompileerd, maar kan zijn opgenomen binnen een wel afzonderlijk compileerbaar bibliotheekpakket. Een taak kan niet op zichzelf staan; hij is steeds van zijn 'ouder' afhankelijk en dat is de eenheid, waarbinnen de taak werd gedeclareerd.

In Ada hoeft een taak niet expliciet te worden opgestart. De taak wordt actief, zodra zijn body is verwerkt aan het eind van het declaratiedeel van zijn ouder. Worden er meerdere taken in hetzelfde declaratiedeel gedeclareerd, dan zijn deze taken alle actief aan het eind daarvan. De volgorde van activering ligt daarbij niet vast, deze is onbepaald of nondeterministisch.

De ouder van één of meer taken kan zelf pas beëindigd zijn als al zijn taken beëindigd zijn. Een taak is beëindigd als de laatste instructie van de taakbody is uitgevoerd en als ook alle van deze taak afhankelijke taken zijn beëindigd. In een volgende paragraaf zullen we nog andere wijzen van beëindiging van taken tegenkomen.

Een taakdeclaratie is tweeledig, net als de pakketdeclaratie (zie figuur 16-3). Een taak bestaat uit een taakspecificatie, waarin de interface tussen de taak en andere eenheden wordt beschreven, en een taakbody, bestaande uit uit te voeren instructies. Evenals bij pakket en subprogramma het geval is, kunnen beide taakcomponenten op verschillende plaatsen in de programmatekst zijn opgenomen.



Figuur 16-3 Schematische weergave van een Ada-taak.

Taakspecificaties

In een taakspecificatie staat de naam van het taakobject (of taaktype, zoals we later zullen zien), tesamen met de grootheden die voor de gebruiker van de taak zichtbaar zijn. Het specificatiegedeelte definieert de communicatiekanalen met andere taken. Voor de eerder als voorbeeld gebruikte taak CONSUMENT kan de specificatie er zo uitzien:

```
task CONSUMENT is
  entry ONTVANG_BERICHT(M : in STRING);
end CONSUMENT;
```

Een entry-declaratie specificeert de communicatie met de buitenwereld en komt qua vorm overeen met een subprogrammaspecificatie. De entry-naam wordt gevolgd door formele parameters, die het type beschrijven van de parameters (berichten) die tijdens een rendez-vous worden ontvangen of verzonden. De formele parameters kunnen de modus **in**, **out** of **in out** hebben, net als bij subprogramma's en deze modus geeft de richting aan van het bericht ten opzichte van de taak, waarbij de entry-declaratie behoort.

Een taak-entry kan worden aangeroepen vanaf elke plaats, waarvandaan ook een subprogramma-aanroep toegelaten is. Dus vanuit een subprogramma, vanuit een taak, een pakketbody (ook vanuit het hoofdprogramma), of vanuit een blok. Het aanroepen door een taak van zijn eigen entry is dus ook mogelijk, hoewel weinig zinvol: als een taak met zichzelf probeert te communiceren kan een situatie van 'deadlock' (= dodelijke omarming) ontstaan, waarin de taak op een rendez-vous met zichzelf blijft wachten.

Deadlock is in het algemeen de situatie, waarin een taak wacht op een hulpmiddel dat nooit beschikbaar kan komen.

Bekijk de volgende, wat ingewikkelder taakspecificatie eens:

```
task BESCHERMDE_STACK is
  pragma PRIORITY(7);
  entry POP(ELEMENT : out INTEGER);
  entry PUSH(ELEMENT : in INTEGER);
end BESCHERMDE_STACK;
```

Een taakspecificatie kan louter entry-aanroepen bevatten, dit in tegenstelling tot een pakket, dat verschillende soorten grootheden kan exporteren. Bij het gebruiken van de taak moet naar de bedoelde entry worden verwezen met behulp van de punt- of componentselectie. De use-clausule kan bij taken niet worden gebruikt, dus entry-aanroepen moeten altijd door de taaknaam worden voorafgegaan. Voorbeelden:

```
BESCHERMDE_STACK.POP(MIJN_WAARDE);
BESCHERMDE_STACK.PUSH(36);
```

Een entry-aanroep kan wat de vorm betreft niet worden onderscheiden van een subprogramma-aanroep. Ja, het is zelfs mogelijk entries tot procedures te benoemen:

```
procedure BESCHERMD_POP(ELEMENT : out INTEGER) renames
          BESCHERMDE_STACK.POP;
```

Nu kan direct BESCHERMD_POP worden aangeroepen. Omdat **use** niet kan worden gebruikt is de mogelijkheid van **renames** vooral handig als taak- en entry-namen lang zijn en we een kortere en toch betekenisvolle naam aan de entry willen geven.

Een entry-aanroep lijkt weliswaar uiterlijk op een subprogramma-aanroep, maar de semantiek is geheel verschillend. Als meer dan één taak hetzelfde subprogramma kan aanroepen, dan is het mogelijk dat een aantal taken tegelijkertijd een bepaald subprogramma aan het uitvoeren zijn; de subprogrammacode wordt dan *re-entrant* (herbetreedbaar) genoemd (dat wil zeggen: de codeverwerking kan worden onderbroken en later worden hervat). Bij entry-declaraties ligt dit anders: bij elke entry hoort impliciet een wachtrij. Als een aantal taken dezelfde entry aanroepen, dan krijgt maar één taak (en wel de taak die de entry het eerst aanriep en dus vooraan in de wachtrij staat) toestemming tot het rendez-vous. De overige taken moeten op hun beurt wachten in de rij, en wel in volgorde van binnenkomst ('first in first out'), en niet volgens hun eventuele prioriteit. De taken in de wachtrij heten *geblokkeerd*; zij moeten wachten op acceptatie van het rendez-vous door hun partner-taak. Zouden twee taken een entry op exact hetzelfde moment aanroepen, dan is de keuze tussen de twee taken willekeurig. Dit kan vooral voorkomen als er sprake is van een 'time-out': een bepaalde tevoren vastgestelde

maximale wachttijd is verlopen en alle wachtende taken zijn in principe tegelijkertijd aan de beurt.

Nu nog een opmerking over de pragma PRIORITY. Bij elke taak, en dus ook bij het hoofdprogramma, behoort een statische prioriteit die de urgentie van de taak aangeeft. De pragma accepteert als invoerparameter een geheeltallige waarde van het type PRIORITY (opgenomen in het pakket SYSTEM), met een implementatie-afhankelijk waardenbereik. Een hogere waarde betekent een hogere urgentiegraad. Als PRIORITY voor een bepaalde taak niet expliciet wordt gespecificeerd, dan kent de computer zelf een waarde toe. De prioriteit heeft geen invloed op de volgorde van behandeling van taken in de wachtrij; deze is steeds 'first in first out', dus 'wie het eerst komt, het eerst maalt'. De prioriteit is echter wel beslissend als meer dan één taak precies tegelijk aan de beurt is.

Een taak is steeds in één van de volgende toestanden:

- in bewerking (er is een processor mee bezig)
- gereed (niet geblokkkeerd en wachtend op verwerking)
- geblokkeerd (uitgesteld of wachtend op een rendez-vous)
- beëindigd (niet of niet meer actief)

PRIORITY geeft ondersteuning bij het toewijzen van verwerkingshulpmiddelen, zoals processoren of geheugenruimte, aan parallelle taken in het geval dat er meer taken aan de beurt voor verwerking zijn dan met de gegeven middelen tegelijkertijd verwerkt kunnen worden. Als dus twee of meer taken met verschillende prioriteit in de toestand 'gereed' zijn, dan wordt het eerst de taak met de hoogste prioriteit voor verdere verwerking gekozen. Als alle taken gelijke, of niet gedefinieerde prioriteit hebben, dan is de keuze onbepaald. In dit laatste geval moet het toewijzingsalgoritme 'fair', dus eerlijk, zijn: geen enkele taak mag 'verhongeren' omdat hij nooit voor verwerking aan de beurt blijkt te komen.

PRIORITY moet alleen gebruikt worden om de relatieve urgentie tussen taken aan te geven. Het getuigt van een slechte programmeerstijl om deze pragma voor synchronisatie van taken te gebruiken, omdat de precieze betekenis van taakprioriteiten machine-afhankelijk is.

Een taak hoeft helemaal geen entries te bezitten, bijvoorbeeld:

```
task PRODUCENT;
```

Deze taakdefinitie heeft geen zichtbare communicatiekanalen; het hoofdprogramma is een voorbeeld van een dergelijke taak. Deze taken zou men 'leidinggevende' taken kunnen noemen (Engels: actor tasks): zij verlenen geen diensten aan andere eenheden maar zijn zelf actief. Ook een actor-taak kan natuurlijk met andere taken, waarvan de entries zichtbaar zijn, communiceren.

De communicatie tussen taken in Ada is asymmetrisch. Het ene uiterste is de actortaak zonder entries; het andere uiterste is de passieve 'dienende' taak (Engels: server task). De servertaak heeft entries, maar roept zelf geen andere taken aan. Ook alle varianten tussen deze twee extremen in, zijn in Ada mogelijk. Dat wil zeggen dat er taken mogelijk zijn, die zowel de entries van andere taken aanroepen, als ook zelf weer andere eenheden bedienen.

De asymmetrie in de communicatie blijkt ook uit de zichtbaarheid van actor/server taken. Een zendende taak moet de naam van de ontvangende taak kennen, maar een ontvangende taak hoeft niet de naam van zendende taken te kennen. Als we nu denken aan de in Ada geldende regel, dat objecten gedeclareerd moeten zijn, voordat zij gebruikt kunnen worden, dan lijkt het onmogelijk dat twee taken elkaar wederzijds kunnen aanroepen. Aan de hand van het volgende voorbeeld zullen we laten zien dat dit wel degelijk mogelijk is:

```
task EERSTE_TAAK is
  entry BEDIENING;
end EERSTE_TAAK;

task TWEEDE_TAAK is
  entry BEDIENING;
end TWEEDE_TAAK;

task body EERSTE_TAAK is . . .
task body TWEEDE_TAAK is . . .
```

Omdat de taakspecificatie en de taakbody gescheiden in de programmatekst kunnen voorkomen, is het mogelijk beide taken eerst te declareren. Nu zijn de entries in beide taken voor de wederzijdse bodies zichtbaar. De body van EERSTE_TAAK kan TWEEDE_TAAK. BEDIENING aanroepen en omgekeerd.

We besluiten deze behandeling van de taak-entry met te wijzen op de mogelijkheid van het creëren van *entry-families*. Zo'n familie bestaat uit een stel gelijkwaardige entries, elk voorzien van een index, zoals bij array-variabelen. Een voorbeeld:

```
type BELANG is (GERING,MIDDELMATIG,HOOG);
task BERICHT is
  entry HAAL(BELANG)(M : out BERICHT_TYPE);
  entry ZET  (BELANG)(M : in  BERICHT_TYPE);
end BERICHT;
```

We kunnen nu naar entries verwijzen, zoals:

```
BERICHT.HAAL(HOOG)(UW_BERICHT);
BERICHT.ZET(LAAG)(MIJN_BERICHT);
```

In een der volgende paragrafen zullen we laten zien hoe entry-families gebruikt worden bij toewijzing van taken ('scheduling').

Tot nu toe gaven we alleen voorbeelden van declaraties van taakobjecten, zoals:

```
task TERMINAL_DRIVER is
  entry HAAL(C : out CHARACTER);
  entry ZET  (C : in  CHARACTER);
end TERMINAL_DRIVER;
```

Deze declaratie is equivalent met:

```
task type NAAMLOOS_TYPE is
  entry HAAL(C : out CHARACTER);
  entry ZET  (C : in  CHARACTER);
end NAAMLOOS_TYPE;

TERMINAL_DRIVER : NAAMLOOS_TYPE;
```

De naam NAAMLOOS_TYPE heeft betrekking op een type, dat voor de gebruiker niet zichtbaar is. Zoals ook geldt voor abstracte datatypen, definieert een *taaktype* een blauwdruk, die kan worden gebruikt om een aantal verschillende taakobjecten te creëren. Een taaktype is **limited private**; er kan dus geen waarde aan worden toegekend en ook tests op gelijkheid of ongelijkheid zijn niet mogelijk. Gebruik als in-parameter voor een subprogramma of taak-entry is wel toegestaan.

Op objecten van het type taak zijn als operaties alleen taak-entries mogelijk, zoals die in de typespecificatie werden gedefinieerd. Met behulp van taaktypen kunnen taakobjecten worden gedeclareerd:

```
task type HULPMIDDEL is
  entry GEEF_VRIJ;
  entry GEBRUIK;
end HULPMIDDEL;

BUFFER : HULPMIDDEL;
SEGMENT : array(1 .. 100) of HULPMIDDEL;
```

Nu kan er verwezen worden naar entries van specifieke objecten:

```
BUFFER.GEEF_VRIJ;
SEGMENT(7).GEBRUIK;
```

In dit laatste voorbeeld wordt van een familie van taken gebruik gemaakt en dit is niet hetzelfde als het gebruiken van een familie van entries. Bij een familie van entries wordt maar één taak gedefinieerd met een aantal verschillende entries (communicatiekanalen). Het object SEGMENT bestaat uit honderd verschillende taken, elk met entries GEEF_VRIJ en GEBRUIK.

Taaktypes kunnen ook in andere declaraties worden gebruikt:

```
type BESCHERMDE_DATA is
  record
    ELEMENT : INTEGER;
    SLEUTEL : HULPMIDDEL;
  end record;
MIJN_DATA : BESCHERMDE_DATA;
type HEAP is access HULPMIDDEL;
```

Door de declaratie van MIJN_DATA wordt één taak geactiveerd. We kunnen entries aanroepen, als:

```
MIJN_DATA.SLEUTEL.GEEF_VRIJ;
MIJN_DATA.SLEUTEL.GEBRUIK;
```

Als tevoren niet bekend is hoeveel taken er precies nodig zijn, dan kan het verbinden van een access-type aan een taaktype zinvol zijn, zoals hierboven bij de definitie van HEAP gebeurde. (Een 'heap' is een datastructuur $h_\ell, h_{\ell+1}, \ldots, h_r$ met de volgende ordening:

$$h_i <= h_{2i} \quad \text{en} \quad h_i <= h_{2i+1}$$

voor $i = \ell, \ldots, r/2$.) Nu wordt een object van het type taak geactiveerd, zodra de toewijzing via 'new' plaatsvindt:

```
POOL : HEAP;
  . . .
POOL : new HULPMIDDEL;   -- activering van de taak
```

Entries kunnen nu worden aangeroepen:

```
POOL.GEEF_VRIJ;
POOL.GEBRUIK;
```

We sluiten onze behandeling van taakspecificaties af met het noemen van de standaardattributen, die bij taakobjecten en taaktypen zijn gedefinieerd:

```
T'ADDRESS        -- startadres van de taak
T'CALLABLE       -- TRUE als T niet gereed of beëindigd is
T'SIZE           -- geheugenruimte nodig voor T
T'STORAGE_SIZE   -- geheugenruimte, gereserveerd voor de
                    activering van T
T'TERMINATED     -- TRUE als T niet meer actief is
```

In Appendix D worden deze attributen beschreven.

Taakbodies

Bij iedere taakspecificatie behoort een taakbody, waarin de door de taak uit te voeren acties worden beschreven. Een taakbody ziet er net zo uit als een subprogrammabody en bestaat eveneens uit een declaratiegedeelte, gevolgd door een rij instructies en eventueel een beschrijving, wat te doen bij excepties (een 'exception handler'). Bij het verwerken van de taak wordt eerst het declaratiedeel uitgewerkt en vervolgens wordt begonnen met het uitvoeren van de acties, gespecificeerd in de instructierij. In veel gevallen is sprake van eindeloos actief blijvende taken; de instructierij vormt dan een eindeloze lus. Een eenvoudige bewakingstaak kan bijvoorbeeld aldus worden geformuleerd:

```
task WATER_MONITOR;
task body WATER_MONITOR is
begin
  loop
    if WATERSTAND > MAXIMUM_NIVEAU then
      GEEF_ALARM;
    end if;
    delay 1.0;
  end loop;
end WATER_MONITOR;
```

Deze taak gaat 'eeuwig' door en laat een alarmsignaal horen, telkens als geconstateerd wordt dat de waterstand te hoog is. De delay-instructie maakt dat de taak steeds minstens één seconde wacht ('inslaapt'), en de waarnemingen worden dus gedaan met een frequentie van één Hertz of lager. Het lijkt er trouwens op dat we hier zondigen tegen de regels voor een goede programmeerstijl, omdat we de globale variabelen WATERSTAND en MAXIMUM_NIVEAU van binnen de taak benaderen. Dit is des te bedenkelijker omdat ook andere taken deze grootheden zouden kunnen wijzigen. Op dit probleem van het gebruik van gemeenschappelijke variabelen komen we in een latere paragraaf nog terug.

Als er bij een taak entries zijn gedefinieerd, dan moet de taakbody minstens één accept-instructie voor elke entry bevatten, hoewel de taal dit niet dwingend oplegt. Bekijk de volgende taakspecificaties:

```
task CONSUMENT is
  entry ZEND_BERICHT(M : in STRING);
end CONSUMENT;
```

Dan is dit bijvoorbeeld de taakbody:

```
with LOW_LEVEL_IO;
use LOW_LEVEL_IO;
task body CONSUMENT is
begin
  loop
    accept ZEND_BERICHT(M : in STRING) do
      SEND_CONTROL(MODEM,M);
    end ZEND_BERICHT;
  end loop;
end CONSUMENT;
```

(Het subprogramma SEND_CONTROL is opgenomen in het standaard-bibliotheekpakket LOW_LEVEL_IO, bedoeld voor communicatie met randapparatuur.)

Voor een rendez-vous tussen taken moet aan twee voorwaarden zijn voldaan: een entry-aanroep van buiten de ontvangende taak en een bijbehorende accept-instructie in de taakbody. Als TAAK_1 gereed is voor rendez-vous, voordat zijn partner TAAK_2 gereed is, dan zal TAAK_1 volgens de regels van Ada wachten. Dit wachten wordt in de regel opgelost als inactief wachten op 'inslapen' en niet als actief wachten. Een ingeslapen taak gebruikt geen processortijd en kan alleen door een andere taak gewekt worden. Bij actief wachten blijft de taak zelf opletten of een van zijn partners tot communicatie bereid is. Zodra rendez-vous plaatsvindt, wordt de rij instructies behorend bij de accept-instructie uitgevoerd. Als de bedienende (ontvangende) taak deze instructies heeft verwerkt dan is het rendez-vous voltooid en zetten beide taken hun werkzaamheden parallel voort.

Bij elke entry kunnen in de taakbody één of meer accept-instructies horen. Een accept-instructie moet expliciet in de taakbody vermeld staan en mag niet zijn ingebed binnen een ander subprogramma. De accept-instructie bestaat uit het gereserveerde woord **accept**, gevolgd door de naam van de entry met eventuele indexen (in het geval van een entry-famile) en een eventueel formeel gedeelte. De bij het rendez-vous uit te voeren acties worden vervolgens beschreven binnen een **accept** ... **do** ... **end** clausule. Bij afwezigheid van acties kan het **do** ... **end** deel worden weggelaten. Een voorbeeld:

```
task REGELAAR is
  entry FASE_1;
  entry FASE_2;
  entry FASE_3;
end REGELAAR;
task body REGELAAR is
begin
  accept FASE_1;
  accept FASE_2;
  accept FASE_3;
    AANVANG_STARTPROCEDURE;
  end FASE_3;
end REGELAAR;
```

In bovenstaand voorbeeld wordt aan de communicatie een bepaalde volgorde opgelegd. Na een rendez-vous in FASE_1 volgt dat in FASE_2 en vervolgens FASE_3. Pas dan kan AANVANG_STARTPROCEDURE worden uitgevoerd. De voorafgaande rendez-vous zijn louter voor taaksynchronisatie bedoeld.

Bij elke taak hoort standaard het attribuut COUNT, dat alleen binnen de taakbody, die standaard bij een bepaalde entry hoort, mag worden gebruikt. COUNT telt het aantal aanroepen in een entry-wachtrij en het gebruik van dit attribuut moet met de nodige omzichtigheid gebeuren: COUNT wordt opgehoogd door binnenkomende aanroepen, maar kan weer worden verlaagd als de gespecificeerde tijdslimiet voor een aanroep verlopen is.

16.2 Taakinstructies



Ada werd in de eerste plaats ontworpen voor real-time toepassingen (toepassingen waarin de werkelijke kloktijd een rol speelt). Het is dan ook niet verwonderlijk dat Ada instructies bevat voor het specificeren van gebeurtenissen op bepaalde tijdstippen. Zo bevat de taal een standaardbibliotheekpakket CALENDAR, dat een type TIME exporteert en een voorgedefinieerde functie CLOCK die de kloktijd retourneert. Het pakket STANDARD bevat een voorgedefinieerd type DURATION, waarmee tijden in seconden kunnen worden uitgedrukt (zie ook Appendix C). Een expressie die de waarde van het type DURATION geeft kan in een delay-instructie worden gebruikt:

```
delay 10.0;          -- vertraging van 10 seconden
delay DURATION(VOLGEND_TIJDSTIP - CALENDAR.CLOCK);
                     -- vertraging voor een bepaald tijdsinterval
```

Een delay-instructie stelt verdere uitvoering van de eenheid waarin de instructie voorkomt uit voor minstens het opgegeven tijdsinterval. We kunnen SECONDEN, MINUTEN en UREN definiëren om de leesbaarheid van de delay-instructie te verbeteren:

```
SECONDEN: constant DURATION := 1.0;
MINUTEN : constant DURATION := 60.0;
UREN    : constant DURATION := 3600.0;
. . .
delay 2.0*UREN + 7.0*MINUTEN + 36.0*SECONDEN;
```

Met opzet zeiden we niet dat een delay-instructie een proces met precies de gespecificeerde tijd vertraagt, maar dat het gaat om specificiatie van de minimale vertraging. Immers, als meer dan één taak door dezelfde processor bediend moet worden, dan is het heel goed mogelijk dat het vertraagde proces na verloop van de gespecificeerde

tijdsperiode niet onmiddellijk weer aan de beurt is. In een ingebedde toepassing kan het van belang zijn, dat wel de precieze tijden kunnen worden gespecificeerd, zoals bijvoorbeeld wanneer metingen op vaste tijdstippen moeten worden uitgevoerd.

Dit is een voorbeeld van een onjuiste manier van vertragen:

```
-- een foute algoritme
loop
  -- een of andere tijdsafhankelijke actie
  delay 30.0*SECONDS;
end loop;
```

Verwerking van de lus duurt minstens 30 seconden, vermeerderd met de tijd nodig voor het uitvoeren van de actie binnen de lus, maar vanwege het taaktoewijzingsmechanisme, dat nondeterministische aspecten bevat, èn vanwege het feit dat er meer programma's van dezelfde hulpmiddelen (schijven, printers etcetera) gebruik kunnen willen maken, is de exacte vertragingstijd nooit zeker.

Een iets betere oplossing luidt als volgt: bereken steeds het volgende starttijdstip voor de lus en specificeer een vertraging ter grootte van de tijdsduur van de huidige kloktijd tot aan dit tijdstip. Dit kan dan zo worden geformuleerd:

```
declare
  SECONDEN          : constant DURATION := 1.0;
  TIJD_INTERVAL     : constant DURATION := 30.0*SECONDEN;
  VOLGEND_TIJDSTIP : CALENDAR.TIME := CALENDAR.CLOCK;
begin
  loop
    delay DURATION(VOLGEND_TIJDSTIP - CALENDAR.CLOCK);
    -- acties die in minder dan TIJD_INTERVAL kunnen worden
       uitgevoerd
    VOLGEND_TIJDSTIP := VOLGEND_TIJDSTIP + TIJD_INTERVAL;
  end loop;
end;
```

In hoofdstuk 17 zullen we zien, dat controle van de verwerkingstijd van een lus nog beter kan gebeuren met behulp van tijdsafhankelijke interrupts.

We hebben nu de grondslagen van taakactivering en beëindiging en de semantiek van het taak rendez-vous behandeld. Eenvoudige communicatie bleek mogelijk met behulp van de entry/accept constructie. We noemen deze wijze van communicatie 'eenvoudig', omdat de ene taak bij het rendez-vous punt op de andere moet wachten. Voor veel problemen is deze simpele oplossing echter niet voldoende: met name de mogelijkheid, dat een taak voor een onbeperkte tijd op een rendez-vous moet wachten, moet in veel gevallen uitgesloten worden. Het moet dan mogelijk zijn een tijdslimiet voor de wachttijd op te geven. Ook kan het voorkomen dat een bepaalde taak moet kiezen uit een aantal verschillende entry-aanroepen van gebruikers

die bediend wensen te worden. Voor dit soort gevallen beschikt Ada over een instructieset, die lijkt op de set sequentiële instructies, die we in hoofdstuk 11 behandelden. Deze instructies zijn echter alleen op taken van toepassing.

De wijze van communicatie tussen taken in Ada kan als volgt worden geclassificeerd:

- eenvoudige communicatie
- door ontvangende taak geselcteerd rendez-vous
- door zendende taak geselecteerd rendez-vous

We behandelen deze communicatievormen aan de hand van een simulatie van een loket bij een bank, waar klanten worden bediend. De gedachtengang achter de communicatieprotocollen binnen Ada is geinspireerd door de communicatie tussen mensen en vandaar dat we kiezen voor dit informele voorbeeld. In de laatste paragraaf van dit hoofdstuk zullen we kiezen voor een meer formele benadering en enkele toepassingen op het gebied van ingebedde systemen behandelen.

Een bankloketbediende en een klant vertegenwoordigen twee onafhankelijke taken met enkele synchronisatiepunten. De activiteiten van de klant zullen bestaan uit zaken als wakker worden, ontbijten, naar zijn werk gaan en, misschien, naar de bank gaan om een geldbedrag te storten. De loketbediende zal ook ontwaken, ontbijten en naar zijn werk gaan, maar in verreweg de meeste gevallen niet op dezelfde tijdstippen, noch op dezelfde plaats als de klant. De grootheden klant en bediende komen alleen met elkaar in aanraking als de klant de bank binnengaat voor een transactie. Laten we om te beginnen uitgaan van een zeer toegewijde loketbediende: zodra hij of zij op de bank is wacht de bediende voor onbepaalde tijd op klanten. In Ada kunnen we dit zo formuleren:

```
-- TAAK KLANT
STORT_BEDRAG(ID      => 1273
             BEDRAG => 1.0);
```

```
-- TAAK BEDIENDE
accept STORT_BEDRAG(ID      : in INTEGER;
                    BEDRAG : in FLOAT) do
  REKENING(ID) := REKENING(ID)+BEDRAG;
end STORT_BEDRAG;
```

Dit is een voorbeeld van eenvoudige communicatie in Ada. Als de KLANT als eerste de entry-aanroep bereikt, dan wacht hij totdat de BEDIENDE het bericht accepteert. Bereikt omgekeerd de BEDIENDE de accept-instructie het eerst, dan wacht hij totdat er een KLANT gereed is voor rendez-vous. Zodra KLANT en BEDIENDE beiden bij het ontmoetingspunt zijn aangeland wordt de do ... end clausule in de accept-instructie van de BEDIENDE uitgevoerd.

Als meer dan één klant de entry STORT_BEDRAG aanroept, dan worden zij geplaatst in de, bij deze entry behorende, impliciet gedefinieerde wachtrij in de volgorde, waarin zij de BEDIENDE aanriepen. Als een DIRECTEUR_KLANT (met een hoge prioriteit) na

een GEMIDDELDE_KLANT (met een lagere prioriteit) arriveert, dan wordt toch de GEMIDDELDE_KLANT het eerst geholpen omdat hij het eerste aankwam.

Erg efficiënt werkt de loketbediende niet: als er in de loop van de dag maar een paar klanten arriveren, dan is de bediende voor het grootste deel van de tijd passief aan het wachten op een klant. We kunnen de bediende een tweede taak geven, bijvoorbeeld het aannemen van telefonische overboekingen. Dit kan als volgt in de taak BEDIENDE worden opgenomen:

```
-- TAAK BEDIENDE
loop
  . . .
  select
    accept STORT_BEDRAG(ID : in INTEGER;BEDRAG : in FLOAT) do
      . . .
    end STORT_BEDRAG;
  or
    accept TELEFONISCHE_STORTING(ID : in INTEGER;BEDRAG : in FLOAT) do
      . . .
    end TELEFONISCHE_STORTING;
  end select;
  . . .
end loop;
```

Deze constructie heet in Ada *selectief wachten*. BEDIENDE kan nu kiezen uit twee mogelijke entries of wachten tot zich een entry aandient. Bij de select-instructie inspecteert BEDIENDE de wachttijen van beide entries. Als beide rijen leeg zijn, dan wacht de taak BEDIENDE en zodra er een KLANT is die ofwel STORT_BEDRAG ofwel TELEFONISCHE_STORTING aanroept, dan wordt deze entry geselecteerd. Als in beide wachtrijen entries staan, dan is de keuze van BEDIENDE willekeurig, maar 'fair': beide entries hebben steeds gelijke kans om gekozen te worden.

We willen nu onze BEDIENDE ook nog bezighouden als er zich helemaal geen klanten aandienen. Hij kan bijvoorbeeld wat administratief werk uitvoeren. Dit kunnen we specificeren in een else-clausule:

```
-- TAAK BEDIENDE
loop
  . . .
  select
    accept STORT_BEDRAG(ID : in INTEGER;BEDRAG : in FLOAT) do
      . . .
    end STORT_BEDRAG;
  or
    accept TELEFONISCHE_STORTING(ID : in INTEGER;BEDRAG : in FLOAT) do
      . . .
    end TELEFONISCHE STORTING;
  else
    DOE_ADMINISTRATIE;
  end select;
  . . .
end loop;
```

Dit is *selectief wachten met een else gedeelte*. Als er nu geen onmiddellijk rendez-vous mogelijk is, dan begint de BEDIENDE met de rij instructies in het **else** gedeelte. Deze instructies kunnen niet worden onderbroken; als zich nu klanten aandienen, dan maakt BEDIENDE toch eerst deze instructies af en begint dan opnieuw bij de **loop**-instructie.

We kunnen het geheel nog verfijnen: veronderstel, dat een klant alleen binnen bepaalde tijden een bedrag kan storten en stel dat de tijden voor telefonische stortingen verschillend zijn. We voegen nu enkele *guard-* of *bewakingsinstructies* toe:

```
-- TAAK BEDIENDE
loop
 . . .
  select
    when BANK_OPEN =>
      accept STORT_BEDRAG(ID : in INTEGER;BEDRAG : in FLOAT) do
        . . .
      end STORT_BEDRAG;
  or
    when LIJN_OPEN =>
      accept TELEFONISCHE_STORTING(ID : in INTEGER;BEDRAG : in FLOAT) do
        . . .
      end TELEFONISCHE_STORTING;
  else
    DOE_ADMINISTRATIE;
  end select;
  . . .
end loop;
```

Dit heet *selectie met guards*. Als een alternatief geen guard heeft, of als de logische expressie van de guard de waarde TRUE heeft, dan is het bijbehorend accept-alternatief 'open'; is de expressie FALSE dan is het alternatief 'gesloten'. In dit laatste geval wordt dit alternatief beschouwd als niet bestaand. De toestand van een guard kan natuurlijk met het verloop van de tijd veranderen, zoals in het hier gebruikte voorbeeld. De guards worden echter alleen geëvalueerd aan het begin van de **select**-instructie, waar de beslissing wordt genomen, wat de volgende actie zal worden. Guards kunnen alleen voor **accept**-instructies staan en niet voor **else**-gedeelten. Als de mogelijkheid bestaat, dat alle alternatieven gesloten zijn, dan is een **else**-deel verplicht. Als dit niet wordt opgenomen dan ontstaat de exceptie PROGRAM_ERROR - een ijverige BEDIENDE moet nu eenmaal wat te doen hebben!

We gaan nog een stapje verder. De BEDIENDE is ook maar een mens en heeft af en toe een pauze nodig. Als de BEDIENDE gedurende twee uur geen klant heeft gekregen, neemt hij EVEN_PAUZE:

```
-- TAAK BEDIENDE
loop
  . . .
  select
    when BANK_OPEN =>
      accept STORT_BEDRAG(ID : in INTEGER;BEDRAG : in FLOAT) do
        . . .
      end STORT_BEDRAG;
  or
    when LIJN_OPEN =>
      accept TELEFONISCHE_STORTING(ID : in INTEGER;BEDRAG : in FLOAT) do
        . . .
      end TELEFONISCHE_STORTING;
  or
    delay 2.0*UREN;
    EVEN_PAUZE;
  end select;
  . . .
end loop;
```

Dit heet *selectie met een delay alternatief*. Alleen als BEDIENDE gedurende twee uur niets te doen heeft wordt EVEN_PAUZE uitgevoerd. De rij instructies na zo'n **delay** kan net zoals na een **else**-gedeelte niet worden onderbroken.

De regels van Ada geven aan, dat meer dan één alternatief in een **select**-instructie een **delay**-gedeelte mag hebben, maar dit heeft weinig zin, omdat altijd de instructierij na de kortste **delay** zal worden gekozen. Als een **delay** alternatief is opgenomen, dan mag geen **else**-deel voorkomen. Dit zou immers altijd in plaats van het **delay**-deel worden verkozen. We kunnen dus kiezen uit:

- selectie met 'else'
- selectie met 'delay'
- selectie met 'guards'
- eenvoudige selectie

Zouden we willen, dat onze BEDIENDE alleen op STORT_BEDRAG reageert en hoogstens 30 minuten tevergeefs zit te wachten, dan formuleren we dat zo:

```
-- TAAK BEDIENDE
loop
  . . .
  select
    accept STORT_BEDRAG(ID : in INTEGER;BEDRAG : in FLOAT) do
      . . .
    end STORT_BEDRAG;
  or
    delay 30*MINUTEN;
    EVEN_PAUZE;
  end select;
end loop;
```

We moeten de taak van de BEDIENDE ook kunnen beëindigen. Taakbeëindiging kan op drie manieren gebeuren. Volgens de regels van de taal kan een taak alleen dan normaal worden beëindigd als zijn ouder aan het eind van zijn instructierij is gekomen en als ook alle van de taak afhankelijke taken beëindigd zijn of op beëindiging wachten. Dit kan worden aangegeven met het alternatief **terminate**. Er mag maar één **terminate** per **select**-instructie voorkomen en dit mag niet met een **delay**- of een **else**-instructie gecombineerd worden. Bijvoorbeeld:

```
-- TAAK BEDIENDE
loop
  . . .
  select
    accept STORT_BEDRAG(ID : in INTEGER;BEDRAG : in FLOAT) do
      . . .
    end STORT_BEDRAG;
  or
    terminate;
  end select;
  . . .
end loop;
```

BEDIENDE wordt nu netjes beëindigd, als de taak die BEDIENDE voortbracht (zijn ouder) gereed is voor beëindiging, en als ook alle eventueel van BEDIENDE afhankelijke taken dat zijn. Als algemene regel kan gelden, dat taakbeëindiging van bedienende taken ordentelijk wordt afgehandeld via **terminate**.

Een tweede manier van taakbeëindiging is, te voorzien in een entry, die wordt aangeroepen zodra de taak beëindigd moet worden. We kunnen bijvoorbeeld de entry SLUITEN toevoegen aan BEDIENDE en de lus via **exit** verlaten als dit bericht wordt geaccepteerd. In het uiterste geval, als een taak om een of andere reden op hol is geslagen, kan de taak beëindigd worden via een **abort**-instructie (dit is de derde manier):

```
abort BEDIENDE;
```

Een taak kan op deze abnormale manier, tesamen met al zijn nakomelingen worden 'afgemaakt', vanaf elke plaats waar de taak zichtbaar is. Dit is een nogal drastische ingreep en deze moet alleen worden uitgevoerd als alle andere middelen te kort schieten.

Een wat zachtaardiger taakbeëindiging bestaat uit een rendezvous met een entry SLUITEN gevolgd door een **delay**- en een **abort**-instructie (dit laatste om er zeker van te zijn dat de taak inderdaad wordt beëindigd):

```
SLUITEN;
delay 30*SECONDEN;
abort BEDIENDE;
```

We kondigen hier de bediende aan, dat we zijn taak gaan beëindigen, wachten dan enige tijd en voeren dan pas de beëindiging uit. Dit wordt wel genoemd: de taak zijn laatste wens laten uitspreken.

Tot nu toe keken we alleen naar selectief rendez-vous voor de bedienende taak; dezelfde opties zijn mogelijk voor de KLANT. Een KLANT wil bijvoorbeeld maar een bepaalde tijd wachten op bediening en vertrekt als hij dan nog niet is bediend:

```
-- TAAK KLANT
. . .
select
  STORT_BEDRAG(ID => 1273, BEDRAG => 1_000.0);
or
  delay 10.0*MINUTEN;
  SCHEP_EEN_LUCHTJE;
end select;
. . .
```

Dit heet een *tijdgebonden entry-aanroep*. Als met de uitvoering van de select-instructie wordt begonnen, dan wordt eerst gewacht op een rendez-vous met BEDIENDE. Wordt de KLANT niet binnen 10 minuten behandeld, dan wordt de entry uit de STORT_BEDRAG wachtrij verwijderd (het attribuut COUNT in de taak BEDIENDE wordt dus verlaagd). Nu worden de instructies na het delay-deel uitgevoerd en de KLANT gaat een luchtje scheppen. Wordt de klant wel binnen 10 minuten bediend, dan gaat de verwerking door bij het einde van de select-instructie.

Hebben we te maken met een erg ongeduldige klant, die geen moment kan wachten, dan kan dat zo worden uitgedrukt:

```
-- TAAK KLANT
. . .
select
  STORT_BEDRAG(ID => 1273, BEDRAG => 1_000.0);
else
  LOOP_WEG;
end select;
. . .
```

Dit heet een *voorwaardelijke entry-aanroep*. Wat de betekenis betreft (qua semantiek) is deze constructie equivalent met een tijdgebonden entry-aanroep met een delay nul, maar Ada biedt deze expliciete vorm om het ontbreken van een tijdsvoorwaarde aan te kunnen geven. (In plaats van een **or** wordt hier een **else** gebruikt.)

Hiermee hebben we de instructies met betrekking tot taken behandeld. We hebben nu een kist vol gereedschappen; in de volgende paragraaf zullen we laten zien hoe we deze kunnen toepassen.

16.3 Toepassingen Van Taken In Ada



Toen we in hoofdstuk 3 subprogramma's behandelden en in hoofdstuk 13 pakketten, zijn we niet alleen ingegaan op de vorm van deze eenheden, maar ook op het correct gebruik ervan. Het werken met taken is in de meeste hogere programmeertalen niet mogelijk, dus zullen we deze mogelijkheid van de taal Ada uitgebreid aan de hand van een aantal voorbeelden illustreren. Subprogramma's vertegenwoordigen bewerkingen en met behulp van pakketten kunnen logisch samenhangende grootheden verzameld worden. Beide constructies weerspiegelen, bij juist gebruik, de structuur van het op te lossen probleem.

Het toepassingsgebied voor taken kan in vier categorieën worden onderverdeeld:

- gelijktijdig uit te voeren acties
- verzenden van berichten
- beheren van gemeenschappelijke hulpmiddelen
- behandelen van interrupts

In de volgende paragrafen zullen deze toepassingen, geïllustreerd met voorbeelden, worden behandeld.

Gelijktijdig uit te voeren acties

Een belangrijke toepassing van taken in Ada is de besturing en uitvoering van parallelle (ook wel 'concurrente') processen. Zoals we al in de inleiding van dit hoofdstuk stelden, zijn er maar weinig hogere programmeertalen, waarin parallellisme kan worden uitgedrukt. In Ada kan dat wel, en wel op een zeer heldere wijze. Bij de uitwerking van een probleemoplossing met behulp van onze objectgerichte ontwerpmethode, kan het geregeld voorkomen dat we acties specificeren, die logisch gezien tegelijkertijd met andere acties kunnen worden uitgevoerd. Dergelijke acties kunnen in taken worden opgenomen.

Laten we bijvoorbeeld eens een computercontrolesysteem voor een auto bekijken. Het systeem controleert onder andere de oliedruk en de koelwatertemperatuur. Het verrichten van deze beide controles kan duidelijk tegelijkertijd gebeuren; het gaat om twee onderling onafhankelijke taken. Veronderstel, dat er een alarmsignaal wordt gegeven als oliedruk of watertemperatuur een bepaalde limietwaarde bereiken. In het declaratiegedeelte van het hoofdprogramma kunnen beide taken worden geïntroduceerd:

```
task OLIEDRUKMETER;
task WATERTEMPERATUURMETER;
task body OLIEDRUKMETER is
  DRUK : FLOAT;
begin
  loop
    MEET(DRUK);
    if DRUK < MIN_DRUK then
      GEEF_ALARM;
    end if;
  end loop;
end OLIEDRUKMETER;

task body WATERTEMPERATUURMETER is
  TEMPERATUUR : FLOAT;
begin
  loop
    MEET(TEMPERATUUR);
    if TEMPERATUUR >MAX_TEMPERATUUR then
      GEEF_ALARM;
    end if;
  end loop;
end WATERTEMPERATUURMETER;
```

Hoewel dit niet blijkt uit dit voorbeeld, zullen MIN_DRUK en MAX_TEMPERATUUR waarschijnlijk constante waarden voorstellen, die in het hoofdprogramma werden gedeclareerd. GEEF_ALARM kan de entry zijn naar een andere taak, maar het kan ook een subprogramma-aanroep zijn, waardoor een hoorbaar en/of zichtbaar alarm afgaat. In de taken worden geen entries gedefinieerd; het zijn dus 'leidinggevende' of 'actor'-taken. Als de gebruikte computer maar één processor heeft, dan zullen beide taken afwisselend door de processor worden bediend (bijvoorbeeld op basis van een zogenaamde 'time-slice' strategie, waarbij beide processen steeds gedurende een bepaald tijdsinterval aan bod komen). De toewijzingsalgoritme moet in ieder geval 'fair' zijn: beide taken moeten geregeld bediend worden. Als er daarentegen evenveel processoren als taken zijn, dan zouden alle taken werkelijk parallel kunnen draaien.

Het gebruik van taken hoeft niet beperkt te blijven tot de hoogste abstractieniveaus van een oplossing; ook binnen algoritmen zijn vaak parallel uitvoerbare componenten te ontdekken. Bij technisch-wetenschappelijk rekenwerk wordt bijvoorbeeld vaak van matrixalgebra gebruik gemaakt. Binnen matrixberekeningen zijn vaak bepaalde operaties parallel op elementen uitvoerbaar. Als voorbeeld geven we de vermenigvuldiging van een matrix met een vector:

$$\begin{vmatrix} X(1,1) & X(1,2) & \ldots & X(1,M) \\ \vdots & \vdots & & \vdots \\ X(N,1) & X(N,2) & \ldots & X(N,M) \end{vmatrix} * \begin{vmatrix} U(1) \\ \vdots \\ U(M) \end{vmatrix}$$

Het resultaat is:

```
|X(1,1)*U(1) + X(1,2)*U(2) + ... + X(1,M)*U(M)|
|                      :                       |
|                      :                       |
|X(N,1)*U(1) + X(N,2)*U(2) + ... + X(N,M)*U(M)|
```

vermenigvuldiging van een matrix van N rijen en M kolommen met een vector van M elementen geeft als resultaat een vector van M elementen. Onmiddellijk duidelijk is, dat alle elementen van het resultaat parallel kunnen worden berekend.

Het hoofdprogramma verstuurt een rij uit de X-matrix en de vector U naar M taken, met behulp van entry-aanroepen ZEND_WAARDEN en krijgt uiteindelijk de elementen van de resultaatvector terug via entry-aanroepen ONTVANG_WAARDE. De M taken PARTIEEL_PRODUCT zijn alle identiek en kunnen als volgt worden geformuleerd:

```
type MATRIX_RIJ is array(INTEGER range <>) of FLOAT;
type POINTER     is access MATRIX_RIJ;
. . .
task type PARTIEEL_PRODUCT is
  entry ZEND_WAARDEN    (V_1,V_2 : in  MATRIX_RIJ);
  entry ONTVANG_WAARDE(P         : out FLOAT);
end PARTIEEL_PRODUCT;
task body  PARTIEEL_PRODUCT is
  PRODUCT  : FLOAT;
  VECTOR_1 : POINTER;
  VECTOR_2 : POINTER;
begin
  accept ZEND_WAARDEN(V_1,V_2 : in MATRIX_RIJ) do
    VECTOR_1 := new MATRIX_RIJ'(V_1);
    VECTOR_2 := new MATRIX_RIJ'(V_2);
  end ZEND_WAARDEN;
  PRODUCT := 0.0;
  for I in VECTOR_1.all'RANGE
    loop
      PRODUCT :=PRODUCT +(VECTOR_1(I) *VECTOR_2(I));
    end loop;
  accept ONTVANG_WAARDE(P : out FLOAT) do
    P := PRODUCT;
  end ONTVANG_WAARDE;
end PARTIEEL_PRODUCT;
```

De taak is onafhankelijk van de dimensies van de matrix en de vector, immers de grenzen van MATRIX_RIJ zijn vrij. Worden de actuele waarden van de parameters ingevuld, dan kan via het attribuut RANGE naar de werkelijke dimensies worden verwezen. De objecten VECTOR_1 en VECTOR_2, waarnaar respectievelijk een rij

van de matrix en de kolomvector verzonden worden, zijn van het access-type. Dit is noodzakelijk, omdat de dimensies pas tijdens de verwerking van het programma bekend zijn. Binnen de lus wordt het partiële produkt berekend en dit wordt vervolgens via ONTVANG_WAARDE door het hoofdprogramma terug ontvangen.

Eenvoudigheidshalve gaan we er vanuit dat de matrix X vierkant is en maximaal de dimensie 10 kan hebben. De matrix X, de vector U en het resultaat P kunnen dan als volgt worden gedeclareerd:

```
type MATRIX is array (1 .. 10) of MATRIX_RIJ(1 .. 10);
X : MATRIX;
U : MATRIX_RIJ(1 .. 10);
P : MATRIX_RIJ(1 .. 10);
```

Een de parallelle produktberekeningsroutine luidt:

```
declare
  PARALLEL_PRODUCT  : array (U'RANGE) of PARTIEEL_PRODUCT;
begin
  for I in U'RANGE
    loop
      PARALLEL_PRODUCT(I).ZEND_WAARDEN(X(I),U);
    end loop;
  for I in U'RANGE
    loop
      PARALLEL_PRODUCT(I).ONTVANG_WAARDE(P(I));
    end loop;
end;
```

De hierboven gedeclareerde familie van taken: PARALLEL_PRODUCT bestaat alleen binnen het bereik van het bijbehorende lokale blok. Naar elke taak wordt een rij van de matrix X gezonden en de gehele vector U. Vervolgens wordt een lus doorlopen, totdat van elke taak een resultaat P is terug ontvangen. Deze methode van berekening is alleen zinvol als ook werkelijk van meer dan één processor gebruik wordt gemaakt: de extra administratie ('overhead'), nodig voor de behandeling van de taken, maakt de methode eerder langzamer dan sneller, als maar van één processor gebruikt wordt gemaakt. Voor het oplossen van stelsels lineaire vergelijkingen zou van deze zelfde techniek gebruik kunnen worden gemaakt.

Versturen van berichten

De volgende toepassingsmogelijkheid van taken is het versturen van berichten. We kunnen berichten willen verzenden naar andere taken, of naar randapparatuur. Er kan bijvoorbeeld een taak worden gedefinieerd voor elk randapparaat (printer, beeldbuisterminal, schijfeenheid) en binnen deze taak worden over te dragen gegevens 'gebufferd'. We willen bijvoorbeeld output sturen naar een printer

en daartoe wordt een zogenaamde buffer gebruikt teneinde de communicatie zo efficiënt mogelijk te laten geschieden. Een taak ZET tekens in dit buffer, zolang dit niet vol is en een tweede taak HAALt tekens uit het buffer en stuurt ze naar de printer, zolang het buffer niet leeg is. Deze werkwijze heet 'spooling' (SPOOL = Simultaneous Peripheral Operation On Line).

We kiezen hier voor een oplossing, die gebruik maakt van het generieke FIFO (First In First Out) pakket, dat we in hoofdstuk 14 ontwikkelden. We veronderstellen verder, dat we met de printer kunnen communiceren via het standaardpakket LOW_LEVEL_IO, dat in hoofdstuk 19 uitgebreid besproken zal worden. De code kan nu als volgt worden geformuleerd:

```
task SPOOL is
  entry ZET(C : in CHARACTER);
end SPOOL;
with LOW_LEVEL_IO,FIFO;
use  LOW_LEVEL_IO,FIFO;
task body SPOOL is
  package CHARACTER_FIFO is new FIFO(ELEMENT => CHARACTER);
  use CHARACTER_FIFO;
  BUFFER : CHARACTER_FIFO.RIJ(LENGTE => 100);
  C      : CHARACTER;
begin
  SCHOON(BUFFER);
  loop
    select
      when not IS_VOL(BUFFER) =>
        accept ZET(C : in CHARACTER) do
          ZET(C, OP =>BUFFER);
        end ZET;
      else
        if not IS_LEEG(BUFFER) then
          HAAL(C, VAN =>BUFFER);
          SEND_CONTROL(PRINTER,C);
        end if;
    end select;
  end loop;
end SPOOL;
```

Het is in Ada niet toegelaten taken te gebruiken als bibliotheekeenheden, daarom wordt de taak vaak in een pakket opgenomen. Het pakket kan vervolgens zichtbaar worden gemaakt via een with-clausule. De pakketspecificatie voor SPOOL kan er bijvoorbeeld zo uitzien:

```
package SPOOLED_PRINT is
  procedure ZET(C : in CHARACTER);
end SPOOLED_PRINT;
```

En de pakketbody wordt:

```
with LOW_LEVEL_IO,FIFO;
use  LOW_LEVEL_IO,FIFO;
package body SPOOLED_PRINT is
  task SPOOL is
    entry ZET(C : in CHARACTER);
  end SPOOL;
  task body SPOOL is
    package CHARACTER_FIFO is new FIFO(ELEMENT => CHARACTER);
    use CHARACTER_FIFO;
    BUFFER : CHARACTER_FIFO.RIJ(LENGTE => 100);
    C      : CHARACTER;
  begin
    SCHOON(BUFFER);
    loop
      select
        when not IS_VOL(BUFFER) =>
          accept ZET(C : in CHARACTER) do
            ZET(C, OP => BUFFER);
          end ZET;
      else
        if not IS_LEEG(BUFFER) then
          HAAL(C, VAN => BUFFER);
          SEND_CONTROL(PRINTER,C);
        end if;
      end select;
    end loop;
  end SPOOL;

  procedure ZET(C : in CHARACTER) is
  begin
    SPOOL.ZET(C);
  end ZET;
end SPOOLED_PRINT;
```

De entry SPOOL_ZET wordt aangeroepen binnen een procedure ZET. Dat hier ZET als een taak is geïmplementeerd blijft dus voor de gebruiker verborgen.

Nu een wat ingewikkelder voorbeeld van het verzenden van berichten, waarbij taken met taken communiceren. Een taak, die zowel berichten verzendt als ontvangt, is zowel leidinggevend als dienend (actor en server) en roept dus entries van andere taken aan en heeft zelf ook entries. In het nu volgende voorbeeld veronderstellen we dat berichten van het type BERICHT zijn en dat elke zendende taak kan kiezen uit drie verschillende bestemmingen. De taak kan daarbij een prioriteit aan het bericht toekennen, waarbij het bericht met de hoogste prioriteit het eerst behandeld wordt. We hebben dus de volgende grootheden nodig:

```
type PRIORITEIT is (LAAG,NORMAAL,HOOG);
type PLAATS     is range 1 .. 3;
--
task BESTEMMING_1 is
  entry ZEND(M : in BERICHT);
end BESTEMMING_1;
--
task BESTEMMING_2 is
  entry ZEND(M : in BERICHT);
end BESTEMMING_2;
--
task BESTEMMING_3 is
  entry ZEND(M : in BERICHT);
end BESTEMMING_3;
```

We veronderstellen hier, dat elk van de drie bestemmingen het bericht op een andere manier verwerkt. Zouden de processen identiek zijn, dan zou een takenfamilie kunnen worden gebruikt.

Het verzenden van berichten gaat bij elke prioriteit op dezelfde manier; hier kan dus van een takenfamilie gebruik worden gemaakt:

```
task VERZEND is
  entry BERICHT_PRIORITEIT(PRIORITEIT)
                              (M : in BERICHT;NAAR : in PLAATS);
end VERZEND;
```

Door de gekozen namen worden aanroepen zeer leesbaar:

```
VERZEND.BERICHT_PRIORITEIT(NORMAAL)(MIJN_BERICHT,
                                    NAAR => 1);
VERZEND.BERICHT_PRIORITEIT(LAAG)(UW_BERICHT,
                                 NAAR => 3);
```

Nog beter was geweest, PLAATS waarden te geven via een enumeratietype, in plaats van door nummering.

De taakbody bestaat uit een algoritme, die een keuze doet uit de mogelijke entries; hiertoe gebruiken we de select-instructie. Guards worden daarbij gebruikt om met de prioriteiten rekening te houden. De algoritme voor de selectie van de bestemming is dezelfde voor elke prioriteit en dit formuleren we nu eerst:

```
procedure KIES_BESTEMMING(M : in BERICHT;NAAR : in PLAATS) is
begin
  case NAAR is
    when 1 => BESTEMMING_1.ZEND(M);
    when 2 => BESTEMMING_2.ZEND(M);
    when 3 => BESTEMMING_3.ZEND(M);
  end case;
end KIES_BESTEMMING;
```

Als we er van uitgaan dat KIES_BESTEMMING voor VERZEND zichtbaar is, dan kan de body van VERZEND als volgt luiden:

```
task body VERZEND is
begin
  loop
    select
      accept BERICHT_PRIORITEIT(HOOG)(M : in BERICHT; NAAR : in PLAATS) do
        KIES_BESTEMMING(M,NAAR);
      end BERICHT_PRIORITEIT;
    or
      when BERICHT_PRIORITEIT(HOOG)'COUNT = 0 =>
      accept BERICHT_PRIORITEIT(NORMAAL)(M : in BERICHT;NAAR : in PLAATS) do
        KIES_BESTEMMING(M,NAAR);
      end BERICHT_PRIORITEIT;
    or
      when BERICHT_PRIORITEIT(HOOG)'COUNT = 0 and
           BERICHT_PRIORITEIT(NORMAAL)'COUNT = 0 =>
      accept BERICHT_PRIORITEIT(LAAG)(M : in BERICHT; NAAR : in PLAATS) do
        KIES_(BESTEMMING(M,NAAR);
      end BERICHT_PRIORITEIT;
    end select;
  end loop;
end VERZEND;
```

We maakten gebruik van het COUNT attrbuut om ervoor te zorgen dat er geen entries met hogere prioriteit in de wachtrij staan.

Beheer van hulpmiddelen

Als een probleem wordt onderverdeeld naar een aantal functies en als vervolgens subprogramma's worden ontwikkeld om die functies te implementeren, dan getuigt het in de regel van een slechte programmeerstijl als daarbij van globale variabelen gebruik wordt gemaakt. Dit verhoogt de graad van koppeling tussen de modulen en maakt het systeem minder betrouwbaar en moeilijker onderhoudbaar (zie hoofdstuk 4). Beter is het, alleen die gegevens binnen een module bekend te laten zijn, die ook daadwerkelijk binnen die module nodig zijn. Bij het gebruik van gegevens door taken ligt dit anders: het kan voorkomen, dat twee of meer taken van dezelfde gegevens gebruik moeten maken. Als een aantal taken een dergelijk gegeven proberen te lezen, terwijl tegelijkertijd een andere taak dit gegeven wijzigt, dan kunnen er problemen ontstaan. De veiligste oplossing is, ervoor te zorgen dat maar één taak op ieder tijdstip een variabele kan benaderen en wijzigen.

Als twee of meer taken een globale variabele moeten kunnen benaderen, dan kan eventueel het volgende worden toegepast:

```
pragma SHARED(variabele_naam);
```

Dit pragma kan op een bepaald object worden toegepast, bijvoorbeeld:

```
pragma SHARED(INDEX)
```

Om een waarde aan INDEX te kunnen geven, of deze waarde te kunnen gebruiken, moet het pragma worden toegepast. De variabele is niet beschermd tegen gelijktijdige benadering door meer dan één taak, dus het is de verantwoordelijkheid van de programmeur, hiertegen te waken. Het SHARED pragma biedt geen enkele mogelijkheid tot taaksynchronisatie, maar moet worden toegepast voor correcte adressering van gemeenschappelijke variabelen, omdat een optimaliserende compiler de lokatie van variabelen kan wijzigen.

Een taak kan ook gebruikt worden om op veilige wijze een bepaald hulpmiddel in gebruik te nemen en weer vrij te geven. Als bijvoorbeeld een database element moet worden gewijzigd, dan kunnen we ervoor zorgen dat op elk moment steeds hoogstens één taak een bepaald record kan gebruiken:

```
GEBRUIK_HULPMIDDEL;
-- wijzig nu de gegevens
GEEF_HULPMIDDEL_VRIJ;
```

De entries GEBRUIK_HULPMIDDEL en GEEF_HULPMIDDEL_VRIJ dienen als semafoor. Als een bepaalde taak het hulpmiddel in gebruik heeft, dan zullen de andere taken bij de entry GEBRUIK_HULPMIDDEL wachten totdat de taak, die zich in het kritieke gebied bevindt, dit verlaat via GEEF_HULPMIDDEL_VRIJ. De semafoor kan als volgt worden geïmplementeerd:

```
task SEMAFOOR is
  entry GEBRUIK_HULPMIDDEL;
  entry GEEF_HULPMIDDEL_VRIJ;
end SEMAFOOR;
task body SEMAFOOR is
  IN_GEBRUIK : BOOLEAN := FALSE;
begin
  loop
    select
      when not IN_GEBRUIK =>
        accept GEBRUIK_HULPMIDDEL do
          IN_GEBRUIK := TRUE;
        end GEBRUIK_HULPMIDDEL;
    or
      when IN_GEBRUIK =>
        accept GEEF_HULPMIDDEL_VRIJ do
          IN_GEBRUIK := FALSE;
        end GEEF_HULPMIDDEL_VRIJ;
    end select;
  end loop;
end SEMAFOOR;
```

Als de benadering van een aantal data-objecten beschermd moet worden, dan is een betere oplossing de creatie van een taaktype voor het beheer van hulpmiddelen. Zoals:

```
task type HULPMIDDEL is
  entry GEBRUIK;
  entry GEEF_VRIJ;
end HULPMIDDEL;
```

De body van dit taaktype kan hetzelfde zijn als de body van SEMAFOOR. Voor de gemeenschappelijke grootheden van een aantal taken kunnen we een record creëren:

```
type BESCHERMDE_DATA is
  record
    D : DATA;
    H : HULPMIDDEL;
  end record;
```

Een taaktype is limited private, dus objecten van het type BESCHERMDE_DATA kunnen niet worden gekopieerd en er kan geen waarde aan worden toegekend. Daaryoe gebruiken we de volgende algoritme:

```
GEMEENSCHAPPELIJKE_DATA : BESCHERMDE_DATA;
. . .
GEMEENSCHAPPELIJKE_DATA.H.GEBRUIK;
-- hier kan GEMEENSCHAPPELIJKE_DATA.D worden gewijzigd
GEMEENSCHAPPELIJKE_DATA.H.GEEF_VRIJ;
```

Om redenen van efficiëntie kan het nog steeds nodig zijn te communiceren via globale data. Een dergelijke communicatiewijze is niet gesynchroniseerd, en slecht beveiligd. De kans is dus groot, dat een niet correct werkende algoritme wordt geproduceerd. De beste oplossing is in dit geval de gemeenschappelijke data binnen een taak op te nemen:

```
task BESCHERMD_ITEM is
  entry ZET(I : in ITEM);
  entry HAAL(I : out ITEM);
end BESCHERMD_ITEM;
task body BESCHERMD_ITEM is
  LOKAAL_ITEM : ITEM;
begin
  loop
    select
      accept ZET(I : in ITEM) do
        LOKAAL_ITEM := I;
      end ZET;
```

```
      or
        accept HAAL(I : out ITEM) do
          I := LOKAAL_ITEM;
        end HAAL;
      end select;
    end loop;
  end BESCHERMD_ITEM;
```

Nu kunnen er een aantal taken zijn, die tegelijkertijd trachten het object te ZETten of de waarde van het object via HAAL op te vragen, maar gelijktijdige toegang wordt door de taak BESCHERMD-_ITEM voorkomen. De taak kan zo worden aangepast, dat ook meer dan één taak het object kan lezen of naar het object kan schrijven; de uitwerking hiervan laten wij echter aan de lezer over.

Interrupts

Binnen ingebedde systemen moet vaak gereageerd worden op asynchrone gebeurtenissen, die door middel van een hardware of software interrupt worden gesignaleerd. In de meeste andere hogere programmeertalen kunnen interrupts alleen worden opgevangen door middel van assembler subroutines. In Ada kunnen interrupts (onderbrekingen van de procesgang) worden behandeld als entry-aanroepen van taken. Als we veronderstellen dat de gebruikte computer naar een bepaald hardware adres springt als de processor een interrupt ontvangt, kan er bijvoorbeeld op de volgende manier een taak aan een interrupt worden verbonden:

```
task SPANNINGS_UITVAL is
  entry FAIL;
  for FAIL use at 16#1FF#;
end SPANNINGS_UITVAL;
task body SPANNINGS_UITVAL is
begin
  accept FAIL
    -- voer bepaalde acties uit;
end SPANNINGS_UITVAL;
```

Hier wordt het SPANNINGS_UITVAL interrupt verbonden aan de adreslokatie 1FF (hexadecimaal), terwijl de taakbody zelf heel ergens anders in het geheugen kan staan. Springt de processor nu vanwege een interrupt naar deze geheugenlokatie, dan is dat equivalent met een aanroep van de entry FAIL. De taak kan de entry accepteren en werken als een interrupt bedienende routine. De for-clausule is een voorbeeld van een representatiespecificatie, die het mogelijk maakt van machine-afhankelijke faciliteiten gebruik te maken. In het volgende hoofdstuk gaan we verder in op deze faciliteiten.

Oefeningen

1. Subprogramma's worden gebruikt om abstracte acties te omschrijven en pakketten worden gebruikt om abstracte objecten of datatypen te omschrijven. Hoe zou u een pakket classificeren, dat een taak bevat?

2. Het sturen van een pakje via de post vergt minstens drie uitvoerders van taken: de afzender, de postbode en de ontvanger. Schrijf de taakspecificaties voor deze drie grootheden. De afzender kan worden behandeld als een zuivere actor-taak (geen entry-aanroepen), de postbode als een taak met zowel een entry (bijvoorbeeld NEEM_PAKJE_AAN) als een aanroep van een entry van ontvanger. De ontvanger tenslotte is een server-taak en heeft alleen een entry: NEEM_PAKJE_IN_ONTVANGST.

3. Schrijf de bodies voor de drie taken uit opgave 2 en maak daarbij gebruik van eenvoudige communicatie tussen taken. Veronderstel dat het pakje van het type SOORT_PAKJE is.

4. Herschrijf de taakbodies uit opgave 3 en voeg daarbij een bevestiging van ontvangst van de ontvanger naar de afzender toe. Er zullen dus entry-aanroepen naar de afzender en naar de postbode moeten worden toegevoegd.

5. Wijzig nu de bodies uit opgave 4 zó dat de afzender hoogstens vijf dagen wacht op de bevestiging van ontvangst. Veronderstel daarbij dat er een constante DAGEN ter beschikking is, waarmee het aantal dagen kan worden uitgedrukt. Als de bevestiging niet op tijd komt, laat dan de exceptie KLACHT_OVER_BEZORGING ontstaan.

*6. Wijzig de bodies uit opgave 5 nu als volgt: als de postbode het pakje niet binnen één dag kan afleveren aan de eerste ontvanger, dan levert hij het pakje af bij een andere ontvanger. De bevestiging van ontvangst moet de naam bevatten van de ontvanger, die het pakje uiteindelijk ontving.

*7. Voeg terminate alternatieven toe aan alle vier bovengenoemde taken (afzender, postbode en twee mogelijke ontvangers), zodat alle taken netjes worden beëindigd als het pakje is afgeleverd en het ontvangstbewijs is terug ontvangen. Ga er vanuit, dat alle vier taken al zijn gedeclareerd in hetzelfde declaratiegedeelte van een andere programma-eenheid.

*8. De in dit hoofdstuk behandelde taak SPOOL heeft nog een schoonheidsfoutje: als BUFFER leeg is en er worden geen tekens verzonden, dan blijft SPOOL actief wachten ('busy waiting'). Herschrijf de body van SPOOL zó dat de 'busy wait' geëlimineerd wordt.

9. Verander de specificatie van SEMAFOOR zodanig dat deze taak een taaktype wordt. Declareer nu een array van 50 semaforen.

17 EXCEPTIEBEHANDELING EN HULPMIDDELEN OP MACHINENIVEAU

Bij het programmeren in machinetaal is men gedwongen zich te verdiepen in de meest primitieve eigenschappen van de computer; bij het programmeren in een hogere taal is het meestal juist niet mogelijk onder een bepaald abstractieniveau af te dalen. Meestal is dit laatste geen probleem, omdat het programmeren in een hogere programmeertaal met veel minder moeite kan geschieden en dus veel produktiever is dan het programmeren in machinecode. Toch is het soms nodig te verwijzen naar systeemafhankelijke zaken, zoals het adres van een input/outputpoort of de fysieke voorstelling van een datastructuur in het geheugen. Tot voor kort was het in die gevallen nodig, de hogere programmeertaal te combineren met in assembler geschreven subroutines en dit maakte de oplossing gecompliceerd en lastig leesbaar en onderhoudbaar.

In ingebedde computersystemen is betrouwbaarheid in de operatiefase een belangrijke factor: in een softwaresysteem in een satelliet of in een atoomcentrale moeten bij voorkeur geen fouten zitten. Toch kunnen er soms uitzonderlijke situaties optreden, die wij niet kunnen voorkomen. Denk aan storingen in randapparatuur of aan het binnenkomen van onverwacht grote hoeveelheden gegevens. Wanneer precies een dergelijke situatie zal optreden is onvoorspelbaar, maar het is wel mogelijk tevoren aan te geven wat in zo'n situatie moet worden gedaan. Men zou kunnen zeggen, dat de programmeur net als de autorijder defensief en anticiperend te werk moet gaan.

Men zou zich dus een taal wensen, waarin bijzonderheden op machineniveau direct op een hoog niveau kunnen worden aangegeven en die helpt bij het defensief programmeren. Deze beide wensen worden door Ada vervuld en in dit hoofdstuk zullen we laten zien hoe in Ada excepties kunnen worden behandeld en hoe systeemafhankelijke details kunnen worden geformuleerd.



17.1 Excepties

Een *exceptie* is in Ada een gebeurtenis, die het normale verloop van het programma onderbreekt. Die gebeurtenis kan een fout zijn, zoals de voorgedefinieerde NUMERIC_ERROR, of het kan een uitzonderlijke situatie zijn, waarin een speciale actie nodig is, zoals een buffer OVERFLOW. Bij voorkeur dient het programma de exceptie op te vangen; in het ergste geval, wanneer herstellen van de fout niet mogelijk blijkt, moet het programma in ieder geval nog op een nette manier eindigen. Creëren van een exceptie brengt de exceptionele situatie onder de aandacht; de reactie op deze situatie heet de exceptiebehandeling.

Tijdens de uitwerking van ons object-georiënteerde logisch ontwerp zullen we meestal een aantal excepties kunnen onderscheiden, die te maken hebben met de eigenschappen van onze objecten. Denk bijvoorbeeld aan een OVERFLOW exceptie bij het ZETten van elementen in een FIFO-wachtrij. Bij datastructuren op een lager abstractieniveau, zoals arrays of INTEGER objecten, zijn een aantal excepties voorgedefinieerd. In beide gevallen getuigt het van goed vakmanschap om deze excepties te voorzien en in de formulering van de oplossing onder te brengen.

Declareren en creëren van excepties

In Ada kan de gebruiker zelf excepties declareren en ook gebruik maken van een aantal voorgedefinieerde excepties. Hoewel een exceptie geen object is, kan een gebruiker toch overal een exceptie declareren, waar ook een objectdeclaratie mogelijk is (de exceptie kan alleen niet als subprogrammaparameter worden gebruikt). Ook wat de vorm betreft ziet de exceptiedeclaratie er hetzelfde uit als de objectdeclaratie: een namenlijst gevolgd door een dubbele punt en het gereserveerde woord **exception**. Hier volgen een paar voorbeelden:

```
OVERSCHRIJDING_BOVENGRENS,
OVERSCHRIJDING_ONDERGRENS : exception;
PARITEITSFOUT             : exception;
FATALE_FOUT_SCHIJF        : exception;
```

De naam van een door de gebruiker gedefinieerde exceptie heeft hetzelfde bereik (scope) als de objectdeclaratie (zie hoofdstuk 20), maar het effect van de exceptie kan verder reiken dan zijn scope.

De volgende excepties zijn voorgedefinieerd in Ada:

```
CONSTRAINT_ERROR
NUMERIC_ERROR
PROGRAM_ERROR
STORAGE_ERROR
TASKING_ERROR
```

Later in deze paragraaf zullen we de voorwaarden behandelen, waaronder deze excepties optreden.

Een door de gebruiker gedeclareerde exceptie wordt gecreëerd met behulp van een expliciete **raise**-instructie:

```
raise FATALE_FOUT_SCHIJF;
raise OVERSCHRIJDING_BOVENGRENS;
raise;
```

raise is een instructie als iedere andere en kan dus ook overal gebruikt worden, waar een instructie mag voorkomen: binnen een blok, in de body van een subprogramma, van een taak of van een pakket. De regels van Ada zorgen ervoor, dat steeds hoogstens één exceptie geldig is binnen een taak, het hoofdprogramma inbegrepen. Treedt een nieuwe exceptie op, dan vervangt deze de vorige. De **raise**-instructie heeft tot gevolg dat de normale sequentiële verwerking wordt onderbroken en dat de besturing wordt overgedragen aan de exceptiebehandeling. Het laatste voorbeeld (een geïsoleerde **raise**-instructie) kan alleen binnen de exceptiebehandeling voorkomen; de exceptie, die de besturingsoverdracht naar de exceptiebehandeling tot gevolg had, wordt dan opnieuw gecreëerd.

De voorgedefinieerde excepties worden in het pakket STANDARD gedeclareerd (zie Appendix C). De creatie van een dergelijke exceptie gebeurt automatisch door het systeem als de voorwaarden daarvoor zich voordoen; het is echter ook mogelijk een voorgedefinieerde exceptie expliciet te creëren:

```
raise NUMERIC_ERROR;
```

Automatisch ontstaan de voorgedefinieerde excepties als volgt:

- CONSTRAINT_ERROR — Ontstaat als een range, index of discriminantbegrenzing overschreden wordt.
- NUMERIC_ERROR — Ontstaat wanneer een numerieke bewerking leidt tot een waarde buiten het gebruikte waardenbereik.
- PROGRAM_ERROR — Ontstaat wanneer alle alternatieven van een **select**-instructie, die geen **else**-gedeelte heeft, gesloten zijn. Als een poging wordt gedaan, een subprogrammapakket of taak aan te roepen, voordat hun body is verwerkt, of als anderszins een incorrecte conditie wordt gesignaleerd.
- STORAGE_ERROR — Ontstaat indien de dynamische geheugenruimte, die aan een grootheid is toegewezen, wordt overschreden.

- TASKING_ERROR — Ontstaat als een exceptie wordt gecreëerd tijdens de communicatie tussen taken.

Een CONSTRAINT_ERROR kan optreden bij objectdeclaraties en bij type-, subtype-, component- en subprogrammadeclaraties. Verder bij initialisaties, waardetoekenningen en **return**-instructies; bij typeconversies en bij subprogramma- en entry-aanroepen. Ook als we proberen te verwijzen naar een grootheid die binnen de huidige programmacontext niet bekend is (denk aan een object, waarnaar via een access-waarde wordt verwezen terwijl die waarde **null** is), zal de exceptie CONSTRAINT_ERROR optreden.

De exceptie PROGRAM_ERROR treedt bijvoorbeeld op als een foutieve situatie wordt gesignaleerd. Een foutieve situatie (Engels: erroneous condition) is een situatie, waarin het effect van het uitvoeren van een instructie niet voorspelbaar is, zoàls bijvoorbeeld het geval is als men een functie probeert te gebruiken, die geen waarde retourneert.

Met nadruk wordt erop gewezen, dat het enige wat men weet als er een exceptie optreedt, is dát die exceptie is opgetreden; de exceptie is alleen maar een symptoom en de programmeur moet ervoor zorgen dat er een algoritme aanwezig is die correcte actie onderneemt naar aanleiding van dit symptoom.

Het rekening houden met en signaleren van excepties maakt wat extra administratie tijdens de verwerking van het programma noodzakelijk. Voor de begrijpelijkheid, betrouwbaarheid en onderhoudbaarheid is het gebruik van excepties echter zeer bevorderlijk. Een efficiëntere verwerking kan meestal eerder worden bereikt door te zorgen voor een goed taalontwerp, dan door het 'peuteren' op detailniveau, zoals bij het weglaten van exceptiesignalering gebeurt. Is er toch een zwaarwegende reden om dit te doen, dan maakt Ada het onderdrukken van een aantal tijdens de verwerking uitgevoerde controles inderdaad mogelijk met behulp van het pragma SUPPRESS. Dit pragma wordt geplaatst in het declaratiegedeelte van een programma-eenheid of van een blok en noemt eerst de naam van de controle en vervolgens de naam van het type, het object, of de eenheid, waarvoor de controle onderdrukt moet worden. Wordt geen naam meegegeven, dan wordt de controle onderdrukt voor de rest van het declaratiegedeelte. De volgende controlenamen kunnen worden gebruikt:

- *onderdrukken van* CONSTRAINT_ERROR *controle*
  ACCESS_CHECK
  DISCRIMINANT_CHECK
  INDEX_CHECK
  LENGTH_CHECK
  RANGE_CHECK
- *onderdrukken van* NUMERIC_ERROR *controle*
  DIVISION_CHECK
  OVERFLOW_CHECK

- *onderdrukken van* PROGRAM_ERROR *controle* ELABORATION_CHECK
- *onderdrukken van* STORAGE_ERROR *controle* STORAGE_CHECK

Willen we bijvoorbeeld de controle op de grenzen van een type INDEX onderdrukken, dan kan dat als volgt:

```
pragma SUPPRESS(RANGE_CHECK,ON => INDEX);
```

Nogmaals: we bevelen het gebruik van dit pragma niet aan als er geen doorslaggevende reden voor is.

Exceptiebehandeling

Als er een exceptie optreedt, zoals bijvoorbeeld na deling door nul, dan wordt in de meeste programmeertalen de verwerking onderbroken en wordt de besturing teruggegeven aan het operating system. Wil een real-time systeem betrouwbaar zijn, dan is zo'n abrupte beeindiging natuurlijk niet toelaatbaar: de exceptionele situatie moet ondervangen worden. Hiertoe biedt Ada de mogelijkheid van het formuleren van 'exceptiebehandeling' in de vorm van een 'exception handler'. Als binnen een eenheid een exceptie optreedt, dan wordt de besturing overgegeven aan de bijbehorende exception handler. Exception handlers kunnen voorkomen aan het einde van een blok of aan het einde van de body van een subprogramma, pakket of taak.

Wat de vorm betreft komt de exception handler overeen met de case-instructie. Ook hier wordt de when-clausule gebruikt om de actie te beschrijven, die naar aanleiding van een bepaalde exceptie moet worden uitgevoerd:

```
declare                                  -- start van een blok
  LAAG_VLOEISTOFPEIL : exception;
begin
. : .
exception
  when LAAG_VLOEISTOFPEIL =>
    OPEN_KLEP;
    GEEF_ALARMSIGNAAL;
  when NUMERIC_ERROR =>
    raise
  when others =>
    LOG_ONBEKENDE_FOUT;
end;
```

In de handler kunnen alle zichtbare excepties (LAAG_VLOEISTOFPEIL en alle voorgedefinieerde excepties) worden genoemd, tesamen met een rij instructies die bij elk van deze excepties moeten worden uitgevoerd. others behandelt alle overige niet genoemde excepties.

In tegenstelling tot wat men misschien zou denken, wordt de besturing nà behandeling van de exceptie niet teruggegeven aan de plaats waar de exceptie werd veroorzaakt, maar wordt de verwerking vervolgd met de instructies, die na de exception handler beschreven staan. Zou geen exceptie optreden, dan zou vanzelfsprekend geen enkele instructie van de exception handler worden uitgevoerd.

In het voorafgaande veronderstelden we eigenlijk dat elke eenheid zijn eigen exception handler bezit voor het opvangen van alle lokale excepties. Beter uit ontwerptechnisch oogpunt is het meestal ook hier het principe van de abstractieniveaus toe te passen. Als vuistregel kan gelden, dat de exceptie moet worden opgevangen op het laagst mogelijke niveau, waar het programma nog correct op de fout kan reageren.

In een pakket voor matrixberekeningen zou bijvoorbeeld door nul gedeeld kunnen worden tijdens een matrixinversie; lokaal kan dan de exceptie NUMERIC_ERROR worden gegenereerd; vervolgens kan bijvoorbeeld de exceptie IS_SINGULIER worden geëxporteerd naar de aanroepende routine, als reactie op de lokale exceptie.

Soms is het juist beter een exceptie op een zo laag mogelijk niveau af te handelen. Een programma dat output probeert te schrijven naar een seriële Input/Outputpoort moet niet lastig gevallen worden met zaken als reageren op pariteitsfouten; dit moet binnen de I/O drivermodule worden behandeld.

Als niet gereageerd wordt op het ontstaan van een exceptie binnen het blok, waarin deze werd gegenereerd, dan werkt de exceptie door tot aan het niveau, waarop deze wel kan worden afgehandeld. Bekijk bijvoorbeeld eens het volgende programma:

```
procedure MAIN is                           ---------   Subprogramma
begin                                                |
. . .                                                |
  declare                               -| Lokaal    |
    LOKALE_FOUT : exception;             | blok      |
  begin                                  |           |
  . . .                                  |           |
  exception                              |           |
    when LOKALE_FOUT =>                  |           |
      DOE_IETS;                          |           |
    when CONSTRAINT_ERROR =>             |           |
      raise;                             |           |
  end;                                  -|           |
  . . .                                              |
exception;                                           |
  when CONSTRAINT_ERROR =>                           |
    DOE_IETS_ANDERS;                                 |
  when NUMERIC_ERROR =>                              |
    DOE_NOG_WAT;                                     |
end MAIN;                                   ---------
```

De voorgedefinieerde excepties, gedeclareerd in het pakket STANDARD zijn zowel zichtbaar in de subprogrammabody als in het blok. De exceptie LOKALE_FOUT is lokaal ten opzichte van het blok. Ontstaat LOKALE_FOUT dan wordt gereageerd met DOE_IETS en gaat de besturing vervolgens naar het einde van dit blok.

Ontstaat daarentegen de fout CONSTRAINT_ERROR binnen het blok, dan doet de exception handler deze fout via **raise** eenvoudig opnieuw ontstaan en plant deze zich voort naar buiten het blok. Daar wordt de fout afgevangen door DOE_IETS_ANDERS. Als tenslotte de fout NUMERIC_ERROR optreedt binnen het blok, dan is daarin door de lokale exception handler niet voorzien; de exceptie plant zich dan automatisch voort naar het eerste niveau waarop deze fout wel kan worden afgevangen.

Zou de exceptie PROGRAM_ERROR ontstaan binnen het blok of binnen het subprogramma, dan is er geen exception handler. In dat geval wordt de controle teruggegeven aan de omgeving die het programma aanriep, in casu het operating system. In het algemeen mogen blokken elk met of zonder exception handler worden genest. Excepties planten zich voort tot op het niveau, waarop zij behandeld kunnen worden.

Voor subprogramma's, die niet het hoofdprogramma zijn, geldt dezelfde regel. Een voorbeeld:

```
procedure MAIN is                                      ┐ Buitenste
. . .                                                  │ sub-
  type KLEIN is digits 5 range 0.0 .. 10.0;            │ programma
  function INVERSE(I : FLOAT)  ┐ Lokaal                │
    return KLEIN is            │ subprogramma          │
  begin                        │                       │
    return KLEIN(1.0/I);       │                       │
  exception                    │                       │
    when NUMERIC_ERROR =>      │                       │
      return 10.0;             │                       │
  end INVERSE;                 ┘                       │
  . . .                                                │
begin                                                  │
. . .                                                  │
  Y := INVERSE(X);                                     │
  . . .                                                │
exception                                              │
  when CONSTRAINT_ERROR =>                             │
    null;                                              │
end MAIN;                                              ┘
```

Zouden we hier INVERSE aanroepen met de actuele parameter 0.0, dan zou de exceptie NUMERIC_ERROR binnen het subprogramma ontstaan. Omdat er een lokale exception handler is, zou deze worden uitgevoerd en de waarde 10.0 retourneren. Als we daarentegen de INVERSE van 0.0001 zouden proberen te berekenen, dan zou de exceptie CONSTRAINT_ERROR ontstaan binnen het sub-

programma INVERSE. Daar de lokale exception handler deze fout niet afvangt, ontstaat dezelfde exceptie op de plaats van de aanroep van de functie.

Als een exceptie optreedt tijdens de verwerking van het declaratiegedeelte van een subprogramma, dan ontstaat deze op de plaats van de subprogramma-aanroep. Als dit subprogramma tevens het hoofdprogramma is, dan wordt de verwerking daarvan gestopt, zelfs als er wel een exception handler is. Dit is ook wel een redelijke strategie, want als de declaraties fouten bevatten dan is verdere verwerking weinig zinvol.

Exceptiebehandeling bij pakketten en taken gaat als volgt: als het pakket geen bibliotheekeenheid is en als een exceptie, waarvoor geen behandeling is gespecificeerd, optreedt in een pakketbody of tijdens de uitwerking van het declaratiedeel van het pakket, dan plant de fout zich voort naar de eenheid die de pakketdeclaratie bevat. Is het pakket wel een bibliotheekeenheid en er treedt een exceptie op, dan wordt de verwerking gestopt, en wel om dezelfde reden als in de vorige alinea vermeld. Wordt een exceptie gesignaleerd bij de uitwerking van het declaratiedeel van een taak, dan ontstaat de exceptie TASK_ERROR op de plaats waar de taak wordt geactiveerd en de taak wordt als voltooid beschouwd. Ontstaat een exceptie tijdens de verwerking van een taakbody en wordt deze niet opgevangen, dan wordt de taak afgemaakt en wordt de exceptie niet verder voortgeplant.

Ook het geval dat er een exceptie tijdens een rendez-vous tussen twee taken optreedt moet bekeken worden. In overeenstemming met de asymmetrische behandeling van taakcommunicatie door Ada, is vastgesteld dat de exceptie TASK_ERROR dan ontstaat in de aanroepende taak, op de plaats van de entry-aanroep indien de aangeroepen taak zichzelf beëindigt, voordat de entry is geaccepteerd, of als deze niet actief was.

Ook als de aanroepende taak zichzelf op abnormale wijze beëindigt tijdens een rendez-vous, dan ontstaat TASK_ERROR binnen deze taak. De exceptie plant zich dan niet voort naar de aangeroepen taak. Als tenslotte de exceptie ontstaat binnen de accept-instructie, dan wordt verwerking daarvan gestopt en wordt dezelfde exceptie binnen de aanroepende taak gecreëerd; dit gebeurt ook op de plaats van de entry-aanroep.

We gaan nu nader in op het verschijnsel, dat een exceptie zich buiten zijn scope kan voortplanten. Een exceptie, die lokaal binnen een blok wordt benoemd kan geldigheid behouden buiten de eenheid, waarbinnen de exceptie gedeclareerd werd, maar we kunnen buiten die eenheid niet de naam van de exceptie gebruiken. Bekijk eens de volgende situatie:

```
declare                                               ┐ Buitenste
. . .                                                 │ blok
begin                                                 │
  . . .                                               │
  declare                                ┐ Lokaal     │
    LOKALE_EXCEPTIE : exception;         │ blok       │
  begin                                  │            │
  . . .                                  │            │
    raise LOKALE_EXCEPTIE;               │            │
    . . .                                │            │
  end;                                   ┘            │
  . . .                                               │
  exception                                           │
    when NUMERIC_ERROR =>                             │
      DOE_IETS;                                       │
    when others =>                                    │
      DOE_IETS_ANDERS;                                │
end;                                                  ┘
```

LOKALE_EXCEPTIE is alleen binnen het lokale blok bekend, maar als LOKALE_EXCEPTIE ontstaat, dan wordt deze buiten het blok voortgeplant, omdat lokaal geen exception handler aanwezig is. In het buitenste blok is de naam van deze exceptie niet bekend; we bevinden ons immers buiten de scope van LOKALE_EXCEPTIE. De exceptie kan nu alleen met behulp van de **others**-clausule worden afgevangen.

Het toepassen van excepties

Zoals bij iedere andere tot nu toe besproken Ada constructie, zijn er goede en slechte manieren om de exceptie te gebruiken. Een exceptie moet bijvoorbeeld niet verworden tot een soort impliciete goto. Waar het om gaat, is het tevoren in de ontwerpfase onderkennen van de mogelijke fouten die kunnen optreden en het toepassen van excepties om deze fouten te ondervangen.

Bij het ontstaan van een exceptie zijn er verschillende strategieën mogelijk, namelijk [1]:

- Stoppen met de verwerking van een eenheid
- De bewerking opnieuw proberen uit te voeren
- Een andere oplossingsmethode proberen
- De oorzaak van de fout wegnemen

Het is niet aan te bevelen, een exceptie domweg te negeren; er moet een of andere actie worden ondernomen. De eerste strategie (stoppen), is toepasbaar als verdere verwerking van de onderhavige eenheid onmogelijk is. Er kan bijvoorbeeld een fatale fout zijn

opgetreden in een of ander randapparaat, waardoor geen verdere I/O meer kan worden gepleegd. De verwerking kan dan het best gestopt worden, zo mogelijk onder vermelding van de fout.

Excepties kunnen worden geëxporteerd vanuit een pakketspecificatie:

```
package IO_INTERFACE is
  procedure ZET(C : in CHARACTER);
  TIJD_OVERSCHREDEN : exception;
end IO_INTERFACE;
package body IO_INTERFACE is
  task IO_DRIVER is
    entry ZEND(C : in CHARACTER);
  end IO_DRIVER;
  task body IO_DRIVER is
    . . .
  end IO_DRIVER;
  procedure ZET(C : in CHARACTER) is
  begin
    select
      IO_DRIVER.ZEND(C);
    else
      delay 5 * MILLISECONDEN;
      raise TIJD_OVERSCHREDEN;
    end select
  end ZET;
end IO_INTERFACE;
```

We definieerden hier een IO_INTERFACE dat aanvragen voor het ZETten van CHARACTERs in een wachtrij plaatst; de taak binnen het pakket zorgt voor het spoolen van deze gegevens. Het ZETten van een teken komt neer op het aanroepen van de taak via een ander subprogramma. Antwoordt het randapparaat niet binnen een bepaalde tijd, dan veronderstellen we een fatale fout. In dit voorbeeld wordt dan de exceptie TIJD_OVERSCHREDEN gegenereerd en deze wordt naar de plaats waar de subroutine ZET werd aangeroepen geëxporteerd. Omdat we de exceptie in de pakketspecificatie de naam TIJD_OVERSCHREDEN gaven, kunnen gebruikers van het pakket ook een exception handler schrijven waarin de exceptie zo genoemd wordt.

In plaats van het definitief opgeven van een bewerking, kunnen we ook proberen de bewerking nog eens opnieuw uit te voeren (tweede strategie). Technisch kunnen we dat oplossen door een lokaal blok te declareren, dat de desbetreffende algoritme omsluit, en vervolgens dit blok door een lus te omgeven, zodat herhaling kan plaatsvinden. Het volgende stuk programma beoogt bijvoorbeeld een fout bij het invoeren door een gebruiker van een enumeratiewaarde op te vangen:

```
type ANTWOORD is (OP,NEER,LINKS,RECHTS);
GEBRUIKERSAANVRAAG : ANTWOORD;
. . .
loop                                    -- herhaal de bewerking
  begin
    PUT(">");
    GET(GEBRUIKERSAANVRAAG);
    exit;                               -- exit indien geen exceptie
  exception
    when DATA_ERROR =>
      PUT_LINE("Onjuist antwoord;
                    kies uit: OP, NEER, LINKS en RECHTS");
  end;
end loop;
```

Bij binnenkomst van het blok wordt een *prompt* of gereedsignaal gegeven en wordt een GEBRUIKERSAANVRAAG afgewacht. Bij een correcte keuze wordt de lus verlaten. Geeft de gebruiker een incorrect antwoord, dan genereert het I/O pakket (zie ook hoofdstuk 19) een DATA_ERROR. Na het geven van een foutboodschap wordt het blok verlaten, maar omdat dit zich binnen een lus bevindt, wordt opnieuw een prompt gegeven en gewacht op invoer. Dit blijft zich herhalen totdat de gebruiker een correct antwoord intikt.

Een wachtlus voor correcte invoer is in veel gevallen een goede ontwerpoplossing, maar er zijn gevallen voorstelbaar, waarin men een bewerking maar een beperkt aantal malen opnieuw wil proberen - bijvoorbeeld bij communicatie met een randapparaat. We maken nu een variant van het vorige voorbeeld, waarbij maar 5 keer om invoer wordt gevraagd:

```
for I in 1 .. 5
  loop                          -- herhaal de bewerking
    begin                       -- start van de beschermde kode
      PUT(">");
      GET(GEBRUIKERSAANVRAAG);
    exit;                       -- exit als geen exceptie
    exception
    when DATA_ERROR =>
      if I < 5 then
        PUT_LINE("Antwoord alleen: OP, NEER, LINKS of RECHTS");
      else
        PUT_LINE("Nu is het welletjes... Uw antwoord is OP");
        GEBRUIKERSAANVRAAG := OP;
      end if;
    end;
  end loop;
```

Nu wordt dus bijgehouden hoe vaak opnieuw geprobeerd is invoer te plegen en wordt een bijbehorende foutmelding gegeven. In plaats van de fout van de gebruiker te herstellen, door een correct

antwoord te veronderstellen, had de exceptie ook naar de volgende hogere programma-eenheid kunnen worden doorgegeven.

Men kan ook in antwoord op een exceptie proberen een andere oplossing te vinden (derde strategie). In een zeer gevoelig communicatienetwerk kan een zeer hoge graad van betrouwbaarheid vereist zijn; we ontwerpen daarom een aantal redundante communicatietaken. (Redundant: eigenlijk overbodig, maar als extra beveiliging meegegeven.) Als we nu een taak aanroepen en een TASKING_ERROR ontvangen, dan kunnen we een ander communicatiekanaal proberen:

```
begin
  ZEND_BERICHT_VIA_ROUTE_1(HOOGST_BELANGRIJK_BERICHT);
exception
  when TASKING_ERROR =>
    ZEND_BERICHT_VIA_ROUTE_2(HOOGST_BELANGRIJK_BERICHT);
end;
```

We gebruikten weer een blok met een lokale exception handler ter beveiliging van een stuk programma.

Als we een taakfamilie gebruiken, in plaats van enkele afzonderlijke communicerende taken, dan kunnen we hier ook het opnieuw proberen van de bewerking inbouwen:

```
ZEND_BERICHT: array(1 .. 10) of BERICHT_TAAK;
. . .
for I in 1 .. 10
  loop
    begin
      ZEND_BERICHT(I)(HOOGST_BELANGRIJK_BERICHT);
      exit;
    exception
      when TASKING_ERROR =>
        if I < 10 then
         null;
        else
         WAARSCHUW_OPERATEUR;
         raise;
      end if;
    end;
  end loop;
```

Hier proberen we het HOOGST_BELANGRIJK_BERICHT via een van de tien mogelijke taken BERICHT_TAAK te verzenden. Als dit tien keer mislukt dan is het devies: WAARSCHUW_OPERATEUR. Tegelijkertijd wordt ook TASKING_ERROR via **raise** opnieuw gegenereerd en doorgezonden naar de volgende exception handler.

Voor een exception handler binnen een subprogramma zijn alle lokale subprogramma-objecten, met inbegrip van de formele parameters, zichtbaar. Daarvan kan gebruik worden gemaakt om te trachten de oorzaak van een fout te verhelpen en dit is onze vierde

strategie. In een besturingssysteem kan bijvoorbeeld een subprogramma worden aangeroepen om een roer te richten. In zo'n geval is er zeker ook een feedback mechanisme ingebouwd om het effect van een besturingsopdracht te controleren. Een te grote verdraaiing moet worden voorkomen, omdat deze het mechanisme zou kunnen beschadigen. Voordat dit kan gebeuren moet er een exeptie worden gegenereerd:

```
procedure DRAAI_ROER(MATE : in out INTEGER) is
  ROER_GEFORCEERD : exception;
begin
  -- zend van hier een commando naar het roerservomechanisme
exception
  when ROER_GEFORCEERD =>
    MATE := MATE/2;
    if MATE /= 0 then
      DRAAI_ROER(MATE);
    else
      raise
    end
end DRAAI_ROER;
```

De waarschuwing ROER_GEFORCEERD kan ontstaan als we een te grote MATE van verdraaiing proberen uit te voeren. We trachten schade te voorkomen door de verdraaiing met de helft te reduceren en roepen de procedure DRAAI_ROER recursief (binnen zichzelf) aan. MATE kan op den duur naar nul convergeren; in dat geval is kennelijk elke verdraaiing fataal. In dat geval wordt de exceptie via **raise** naar buiten gepropageerd.

17.2 Specificatie Van Representatie



In ingebedde computersystemen zullen we ons vaak ook bezig moeten houden met machine-afhankelijke aspecten. Bij het ontwerp van een Input/Output pakket voor een bepaald apparaat kan bijvoorbeeld verwezen worden naar I/O poorten met bepaalde vaste geheugenadressen. Of we willen bijvoorbeeld de efficiëntie verhogen van het schrijven naar een bepaalde schijfeenheid, door een vaste recordlengte te kiezen en een bepaalde wijze van datacompressie. In geen van deze gevallen hoeven we buiten Ada om te werken; de taal bezit verscheidene constructies, waarmee naar machine-afhankelijke aspecten kan worden verwezen. In deze paragraaf behandelen we de mogelijkheden van Ada voor het weergeven van gegevens op het allerlaagste abstractieniveau, dat wil zeggen, op het niveau dat het dichtst bij de machine staat.

Bij het ontwerpen van computerprogrammatuur is het niet verstandig op het laagste abstractieniveau te beginnen. Veel beter is het, ook uit het oogpunt van overdraagbaarheid en onderhoudbaarheid, om eerst op een hoog abstractieniveau en zoveel mogelijk machine-onafhankelijk te werk te gaan. Bij het uitvoeren van een stapsgewijze verfijning worden dan deze aspecten op het laagste niveau alleen dan uitgewerkt als dit strikt noodzakelijk is. Maar al te vaak zijn we aan het priegelen op microniveau, in plaats van ons te concentreren op het verkrijgen van een correcte macrostructuur. Bij ingebedde systemen komt het trouwens vaak voor, dat de hardware en de software tegelijkertijd nog in het ontwerpstadium zijn. Vaak moet de software worden ontworpen terwijl bijvoorbeeld de hardware interfaces met randapparatuur nog niet bekend zijn. In die gevallen is het zeker noodzakelijk te beginnen op een hoog abstractieniveau, waarin de details later kunnen worden ingevuld.

Pas als we er zeker van zijn, dat ons systeem logisch consistent en correct is, wordt het tijd om ons zorgen te gaan maken over de representatie op machineniveau. Het eindeffect van een programma hangt immers niet af van de voorstellingswijze van de gegevens en in veel gevallen hoeven we ons daar helemaal niet mee bezig te houden; dan kiest de compiler de machinerepresentatie.

Expliciete specificatie van de representatie moet alleen om redenen van efficiënte verwerking worden gebruikt; we moeten dan naar een specifieke machine-eigenschap kunnen verwijzen met behulp van de gebruikelijke Ada-namen. Ook kan dit nodig zijn, als een interface met een ander systeem moet worden beschreven. De machine-eigenschappen kunnen dan in Ada op dezelfde abstracte manier worden behandeld als andere grootheden.

Een representatiespecificatie geeft aan hoe bepaalde grootheden uit de geformuleerde oplossing moeten worden afgebeeld naar de gebruikte computer. Deze specificatie kan voorkomen in het declaratiegedeelte van een programma-eenheid en van taken of pakketten. De representatiespecificatie mag alleen worden gegeven als het bijbehorende type is gedeclareerd en vóórdat objecten van dit type worden gedeclareerd.

Aanwijzingen voor de voorstellingswijze van datastructuren en code optimalisatie kunnen door middel van de volgende pragma's (compileraanwijzingen) worden gegeven:

```
pragma PACK(EEN_TYPE);
pragma OPTIMIZE(TIME);
pragma OPTIMIZE(SPACE);
```

Het eerste pragma PACK geeft aan, dat objecten van het type EEN_TYPE zo voordelig mogelijk moeten worden opgeslagen (denk bijvoorbeeld aan een rij BOOLEAN elementen, waarbij voor elk element maar één bit nodig is). De volgende twee pragma's bevatten een aanwijzing voor de compiler om respectievelijk de verwerkingstijd of de opslagruimte te minimaliseren. Alle voorgedefinieerde pragma's worden in Appendix E in detail beschreven.

Er zijn in Ada vier categorieën representatiespecificaties:

- lengtespecificatie
- enumeratietype representatie
- recordtype representatie
- adresspecificatie

Lengtespecificatie bepaalt de hoeveelheid geheugenopslagruimte die aan een bepaalde grootheid wordt toegekend en ziet er als volgt uit:

```
for attribuut use eenvoudige expressie;
```

Het attribuut specificeert het soort lengtespecificatie en de eenvoudige (dat wil zeggen niet samengestelde) expressie levert een waarde op, waarvan de interpretatie afhangt van het attribuut. De volgende attributen kunnen zinvol worden gebruikt: SIZE, STORAGE_SIZE en SMALL. (Zie Appendix D voor de betekenis van deze attributen.) Met behulp van het SIZE attribuut is het mogelijk een bovengrens aan te geven voor het aantal bits, dat voor objecten van een bepaald type gebruikt mag worden:

```
BITS : constant := 1;
type MIJN_INTEGER is range -100 .. 100
for MIJN_INTEGER'SIZE use 8*BITS;
```

De declaratie van de constante BITS is louter ter verbetering van de leesbaarheid toegevoegd. Elk object van het type MIJN_INTEGER zal nu hoogstens 8 bits in beslag nemen. Men dient wel te bedenken, dat een dergelijke declaratie op verschillende machines verschillende effecten kan hebben voor de op verschillende machines geproduceerde code: de overdraagbaarheid van de programmatuur neemt er dus door af.

Op soortgelijke manier kan de opslagruimte, die voor een verzameling objecten moet worden gereserveerd, worden gespecificeerd en ditzelfde kan ten aanzien van een te activeren taakobject:

```
BYTES       : constant := 8*BITS;
KILO_BYTES  : constant := 1024*BYTES;
type RECORD_POINTER is access BUFFER;
for RECORD_POINTER'STORAGE_SIZE use 100*BYTES;

task BEWAKINGSTAAK is
  . . .
end BEWAKINGSTAAK;
for BEWAKINGSTAAK'STORAGE'SIZE use 3*KILO_BYTES;
```

Op deze manier worden er 100 bytes opslagruimte gereserveerd voor objecten, waarnaar met behulp van RECORD_POINTER verwezen kan worden. De STORAGE_SIZE voor de taak heeft betrekking op de voor activering van de taak te reserveren geheugenopslagruimte (het gaat hierbij om de door de taak te genereren gegevens en niet om de programmacode van de taak).

STORAGE_SIZE wordt gemeten in de opslageenheden van de gebruikte machine en dit heeft betrekking op de woordlengte (zie ook het pakket SYSTEM in Appendix C). De exceptie STORAGE-_ERROR wordt gegenereerd wanneer de gereserveerde opslagruimte overschreden dreigt te worden.

De lengtespecificatie kan ook gebruikt worden om de precisie bij vaste puntvoorstellingen van numerieke gegevens te specificeren:

```
type RADIALEN is delta 0.001 range 0.0 .. 1.0;
for RADIALEN'SMALL use (2.0**(-10));
```

De werkelijke precisie moet hierbij kleiner zijn dan de in de typedeclaratie aangegeven **delta**. Als efficiëntie van berekeningen van groot belang is, kan deze constructie van pas komen.

Bij enumeratietypen kan ook een representatiespecificatie worden gebruikt. Deze specificeert hoe de mogelijke waarden, zoals in de enumeratie opgesomd, intern moeten worden gecodeerd. De specificatie ziet er hetzelfde uit als een lengtespecificatie, maar daarbij komt een aggregaat ter specificatie van de te gebruiken afbeelding:

```
type ANTWOORD is (OP,NEER,LINKS,RECHTS);
for ANTWOORD'SIZE use 4*BITS;
for ANTWOORD use (OP     => 2#0001#,
                  NEER   => 2#0010#,
                  LINKS  => 2#0100#,
                  RECHTS => 2#1000#);
```

OP, NEER, LINKS en RECHTS zijn nu direct vertaald naar bepaalde bitpatronen, door van een binaire getalsvoorstelling gebruik te maken.

Bij een dergelijke representatiespecificatie voor de waarden van een enumeratietype kunnen alleen geheeltallige letterlijke getalsvoorstellingen worden gebruikt. Deze hoeven niet per se opeenvolgend te zijn; de attributen POS, PRED en SUCC blijven desondanks correct werken.

Bovenstaande enumeratietype representatiespecificatie wordt bijvoorbeeld gebruikt voor het benoemen van bitpatronen in Ada. Een systeem kan bijvoorbeeld instructies versturen naar een randapparaat, zoals een schijfbesturingseenheid. Terwille van de leesbaarheid willen we deze instructies graag begrijpelijke namen geven, zelfs als de fabrikant de instructies alleen als bitpatronen heeft gespecificeerd. Het zou bijvoorbeeld om de volgende instructies kunnen gaan:

```
type COMMANDO is (HOME,ZOEK,STAP,PLAATS,LEES,SCHRIJF);
```

(HOME brengt bijvoorbeeld de lees/schrijfkop terug naar een uitgangspositie.)

Elk COMMANDO is in feite een *mnemonic* (= geheugensteuntje) voor een bepaald bitpatroon en de enumeratiespecificatie kan nu als volgt worden geformuleerd:

```
for COMMANDO'SIZE use 6*BITS;
for COMMANDO use (HOME    => 8#00#,
                  ZOEK    => 8#04#,
                  STAP    => 8#06#,
                  PLAATS  => 8#10#,
                  LEES    => 8#50#,
                  SCHRIJF => 8#70#);
```

Plaatsen we nu een van de mogelijke waarden van COMMANDO ergens in het geheugen, dan wordt daar in feite de bijbehorende bitstring weggeschreven. De mogelijkheid om bij de specificatie de getalswaarden in een notatie met een bepaald grondtal aan te geven (binair, octaal), vergroot de leesbaarheid van het geheel. In de volgende paragraaf zullen we zien dat deze zelfde constructie ook gebruikt kan worden om de symbolische assemblerinstructies (mnemonics) van een bepaalde targetmachine (de machine waarop het systeem uiteindelijk moet draaien) weer te geven.

De voorstellingswijze van record datatypen kan op soortgelijke wijze worden gespecificeerd. Op die manier is het mogelijk precies aan te geven hoe en waar records moeten worden opgeslagen. Dit kan bijvoorbeeld worden toegepast ter specificatie van bepaalde adresseerbare machinecomponenten. Een bepaalde Input/Outputpoort met bijbehorende statusbits kan nu als volgt abstract worden voorgesteld:

```
type IO_PORT is
  record
    DATA              : INTEGER range 0 .. 255;
    READY             : BOOLEAN;
    INTERRUPT_ENABLED : BOOLEAN;
  end record;
```

Als we veronderstellen, dat onze computer een woordlengte heeft van 1 byte en dat een IO-poort twee opeenvolgende bytes gebruikt, dan kan de representatie als volgt worden gespecificeerd:

```
WORD : constant := 1*BYTE;
. . .
  for IO_PORT use
    record at mod 2;
      DATA              at 0*WORD range 0 .. 7;
      READY             at 1*WORD range 3 .. 3;
      INTERRUPT_ENABLED at 1*WORD range 7 .. 7;
  end record;
```

De **at mod** constructie wordt gebruikt om de wijze van vulling van het record aan te geven. Het totale record bestaat uit twee woorden, genummerd 0 en 1. DATA begint in het eerste woord en beslaat alle 8 bits, genummerd van 0 t/m 7. READY neemt één bit in beslag en wel bit 3 van het tweede woord en de interruptvlag wordt gevormd door bit 7.

Tenslotte kan in Ada nog de representatie van een adres worden aangegeven. De lokatie van een variabele of een constante kan bijvoorbeeld als volgt worden gespecificeerd:

```
DIGITAAL_ANALOOG_CONVERTER : IO_PORT;
for DIGITAAL_ANALOOG_CONVERTER use at 16#177F6#;
                    -- pakket SYSTEM moet zichtbaar zijn
```

We declareerden een object van het type IO_PORT en specificeerden zijn absolute adreslokatie als 177F6 in hexadecimale getalsvoorstelling. Deze constructie kan ook worden gebruikt om het startadres van een programma-eenheid, zoals een subprogramma, pakket of taak op te geven. Het gaat hier dan om het beginadres van de bij deze eenheid behorende uit te voeren code. Het operating system van een bepaalde machine kan bijvoorbeeld vanaf 8#76# een bepaalde routine hebben staan voor het op een gecontroleerde manier uitzetten van het systeem en deze kan dan als volgt worden beschreven:

```
procedure POWER_DOWN;
for POWER_DOWN use at 8#76#;
                    -- pakket SYSTEM moet zichtbaar zijn
```

Wordt nu POWER_DOWN vanuit een Ada programma aangeroepen, dan wordt een directe aanroep van een stuk machinecode uitgevoerd. (Van deze mogelijkheid moet géén gebruik gemaakt worden voor het aanroepen van programma overlays.)

We geven nog een voorbeeld van het gebruik van de adresspecificatie voor het aangevan van een interrupt binnen een taak:

```
task KOELING_DEFECT is
  entry TEMPERATUUR_INTERRUPT;
  for TEMPERATUUR_INTERRUPT use at 16#3E#;
end KOELING_DEFECT;

task body KOELING_DEFECT is
begin
  accept TEMPERATUUR_INTERRUPT do
    POWER_DOWN;
  end TEMPERATUUR_INTERRUPT;
end KOELING_DEFECT;
```

Als er een hardware interrupt wordt gegeven op lokatie 3E (hex), dan vangt de taak KOELING_DEFECT dit signaal op en roept

vervolgens POWER_DOWN aan. Er mag maar één taak-entry met een bepaald interrupt verbonden worden; zouden we er meer mee verbinden dan wordt het programma als foutief beschouwd.

Tot dusverre hebben we er steeds de nadruk op gelegd, dat bij elk type in Ada ook maar één voorstellingswijze moet behoren. Toch zijn er gevallen, waarin meer dan één voorstellingswijze voor hetzelfde type gewenst is. Om een voorbeeld te geven: veronderstel, we moeten in het veld een groot aantal afstandsbepalingen uitvoeren. Wegens beperkte opslagcapaciteit van onze draagbare apparatuur willen we de metingen zo gecomprimeerd mogelijk opslaan, maar als berekeningen moeten worden uitgevoerd is het niet handig om de gegevens steeds te moeten 'unpacken' en weer te moeten 'packen'. Dit is op te lossen door gebruik te maken van een opslagrepresentatie en van een rekenrepresentatie van dezelfde gegevens.

We kunnen een UNPACKED record en een bijbehorend PACKED record declareren:

```
type UNPACKED is
  record
    . . .
  end record;

type GECOMPRIMEERD is new UNPACKED;
pragma PACK(GECOMPRIMEERD);
```

Er is nu sprake van twee samenhangende typen met verschillende representaties. In hoofdstuk 8 lieten we echter al zien, dat een typeconversie tussen een afgeleid type en zijn voortbrenger is toegestaan:

```
REKENDATA   : UNPACKED;
OPSLAGDATA  : GECOMPRIMEERD;
. . .
REKENDATA   := UNPACKED(OPSLAGDATA);
OPSLAGDATA  := GECOMPRIMEERD(REKENDATA);
```

Door het toekennen van duidelijke namen wordt de hele conversieprocedure buitengewoon leesbaar.

17.3 Systeemafhankelijke Aspecten



Dat vanuit Ada representaties expliciet kunnen worden aangegeven is een zeer waardevolle eigenschap van de taal. Als dit niet mogelijk was, dan zou men buiten de taal om moeten gaan om verwerking op machineniveau te realiseren. Maar Ada gaat nog verder: het is mogelijk direct vanuit Ada naar een aantal machine-afhankelijke

grootheden te verwijzen. Zo bestaat er een pakket SYSTEM, dat een aantal systeemafhankelijke constanten bevat.

Verder is het mogelijk onze specifieke systeemconfiguratie door middel van pragma's aan te geven (zoals SYSTEM en STORAGE-_SIZE) en kunnen we met behulp van attributen (zoals MACHINE-_RADIX en POSITION) naar systeemafhankelijke aspecten verwijzen. Een bepaalde implementatie van Ada mag een eigen verzameling systeempragma's en attributen hebben, maar moet minstens de verzameling, zoals in de Appendices D en E gedefinieerd, bevatten.

Bij een bepaalde toepassing, bijvoorbeeld een subprogramma dat per se binnen een bepaalde zeer korte tijd moet worden uitgevoerd, kan het nodig zijn direct in machinetaal te schrijven. Het is daarbij niet aan te raden direct in machinecode te beginnen; we moeten ons systeem op abstract logisch niveau formuleren en alleen de werkelijke kritieke onderdelen in machinetaal hercoderen.

Ook het schrijven in assemblercode is in Ada mogelijk. De symbolische machine-instructies zetten we in een subprogramma, zonder andere declaraties of instructies, en we definiëren een pakket, dat een record exporteert met de namen (mnemonics) van deze instructies:

```
package MACHINE_CODE is
  BITS   : constant := 1;
  WOORD : constant := 8;
  type OPCODE is (MOV,SUB,ADD);
  for OPCODE'SIZE use 2*BITS;
  for OPCODE use (MOV => 2#11#,
                  SUB => 2#01#,
                  ADD => 2#10#);
  type REGISTER is range 0 .. 7;
  for  REGISTER'SIZE use 3*BITS;
  type INSTRUCTIE is
    record
      COMMANDO     : OPCODE;
      SOURCE       : REGISTER;
      DESTINATION : REGISTER;
    end record;
  for INSTRUCTIE use
    record at mod 1;
      COMMANDO    at 0*WORD range 0 .. 1;
      SOURCE      at 0*WORD range 2 .. 4;
      DESTINATION at 0*WORD range 5 .. 7;
    end record;
end MACHINE_CODE;

with  MACHINE_CODE;
use  MACHINE_CODE;
procedure COPY_3 is
begin
  INSTRUCTIE'(COMMANDO => MOV,SOURCE => 0,DESTINATION => 1);
  INSTRUCTIE'(COMMANDO => MOV,SOURCE => 0,DESTINATION => 2);
  INSTRUCTIE'(COMMANDO => MOV,SOURCE => 0,DESTINATION => 3);
end  COPY_3;
```

Nu kunnen we de machinecodeprocedure aanroepen:

```
COPY_3;
```

Door deze aanroep worden de assembler instructies ook daadwerkelijk uitgevoerd.

Dit voorbeeld is wel wat geïdealiseerd: meestal kent een machine instructies van verschillende vorm en verschillende lengte.

De laatste te behandelen machinegebonden constructie in Ada maakt het mogelijk de typecontrole en verwerkingsregels van de taal gedeeltelijk uit te schakelen. De uitwerking hiervan is buitengewoon machine-afhankelijk en veilig gebruik is geheel onder verantwoordelijkheid van de programmeur. Ada heeft twee generieke subprogramma's om controles uit te schakelen:

```
generic
  type OBJECT  is limited private;
  type NAME    is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);

generic
  type SOURCE is limited private;
  type TARGET is limited private;
function UNCHECKED_CONVERSIONS(S : SOURCE) return TARGET;
```

Als men in een programma van deze mogelijkheden gebruik wil maken, dan moet daarin vermeld worden:

```
with UNCHECKED_DEALLOCATION;
with UNCHECKED_CONVERSION;
```

Zo wordt het gebruik van een niet beveiligde programmeermethode expliciet vermeld. Als van UNCHECKED_DEALLOCATION gebruik wordt gemaakt, dan wordt het weer beschikbaar stellen van niet meer gebruikte geheugenlokaties (garbage collection) niet meer automatisch uitgevoerd, maar kan dit voor een bepaald object rechtstreeks gebeuren door van deze procedure gebruik te maken.

Vanzelfsprekend houdt dit allerlei gevaren in. Met name is er geen beveiliging tegen het verwijderen van een object, waarnaar door andere objecten nog verwezen wordt. Een voorbeeld:

```
type BUFFER is array (1 .. 100) of CHARACTER;
type POINTER is access BUFFER;
procedure GEEF_BUFFER_VRIJ is new
  UNCHECKED_DEALLOCATION(BUFFER,POINTER);
HEAD : POINTER := new BUFFER;
TAIL  : POINTER;
. . .
TAIL := HEAD;
. . .
GEEF_BUFFER_VRIJ(HEAD);
```

Door de laatste instructie wordt het object, waarnaar HEAD verwijst, verwijderd. HEAD verwijst nu naar null, maar TAIL verandert niet van waarde en verwijst nu dus naar een niet bestaand object.

Via UNCHECKED_CONVERSION kunnen we volledig vrij ieder type naar ieder ander type converteren. Dit kan handig zijn als het ooit nodig is twee anders niet bij elkaar passende typen naar elkaar te converteren. Het effect van de functie is, dat de bitstring van de parameter als waarde van de TARGET variabele wordt geretourneerd. Een voorbeeld:

```
type SEGMENT is array (0 .. 1023) of WOORD;
type POINTER is access SEGMENT;
. . .
NIEUW_GEHEUGEN : POINTER;
START_ADRES    : INTEGER;
. . .
function GEHEUGENPOINTER is new
  UNCHECKED_CONVERSION(SOURCE => INTEGER,
                       TARGET => POINTER);
. . .
NIEUW_GEHEUGEN := GEHEUGENPOINTER(START_ADRES);
```

We converteren hier een INTEGER waarde naar een access waarde en daardoor wordt het mogelijk een SEGMENT uit het geheugen toe te wijzen. Ook hier is het volledig voor de verantwoordelijkheid van de programmeur ervoor te zorgen dat een dergelijke conversie niet tot problemen leidt.

Hiermee sluiten we onze bespreking van de mogelijkheden van Ada voor het programmeren op machineniveau af. We zullen nu in staat zijn met behulp van dit gereedschap en met behulp van het eerder behandelde taakbegrip, de wat ingewikkelder real-time problemen, die in de volgende hoofdstukken worden behandeld, op te lossen.

Oefeningen

1. Stel, we beschikken over een subprogramma MEET_SPANNING om voltages te meten. Bij te hoge of te lage waarden genereert het de excepties TE_HOOG_VOLTAGE en TE_LAAG_VOLTAGE. Schrijf een pakket, waarin een subprogramma dat voortdurend MEET_SPANNING aanroept. Als totaal 10 excepties zijn geteld, moet de verwerking worden gestopt en moet de exceptie STROOMSTORING door het pakket worden geëxporteerd.

2. Pas het pakket uit opgave 1 zo aan, dat de exceptie STROOMSTORING alleen ontstaat als de 10 excepties binnen één minuut werden geteld.

3. Schrijf een recordtype declaratie voor een 16 bits programmastatuswoord met een veld dat een prioriteit aangeeft in de bits 0 t/m 3, een vlag die aangeeft of een resultaat al of niet nul was (zero flag) in bit 9, een vlag die aangeeft of een resultaat negatief was in bit 10, een vlag die aangeeft of bij een bewerking een eenheid in de volgende positie moet worden meegenomen (carry flag) in bit 11, en een 'overflow flag' die aangeeft of een resultaat een gegeven waardebereik heeft overschreden in bit 12. Declareer een bijbehorend object en schrijf een adresspecificatie om dit object in geheugenlokatie 8#777_776# te plaatsen.

*4. Formuleer een typedeclaratie voor een array van 32 elementen van het type BINARY. Veronderstel dat BINARY een enumeratietype is met waarden NUL en EEN. Schrijf de representatiespecificaties (niet het pragma) nodig om objecten van dit arraytype in twee 16 bits woorden op te slaan.

18 HET VIERDE ONTWERPPROBLEEM: PROCESBESTURING

Voor ingebedde computersystemen zijn in het algemeen de volgende vier vereisten nodig:

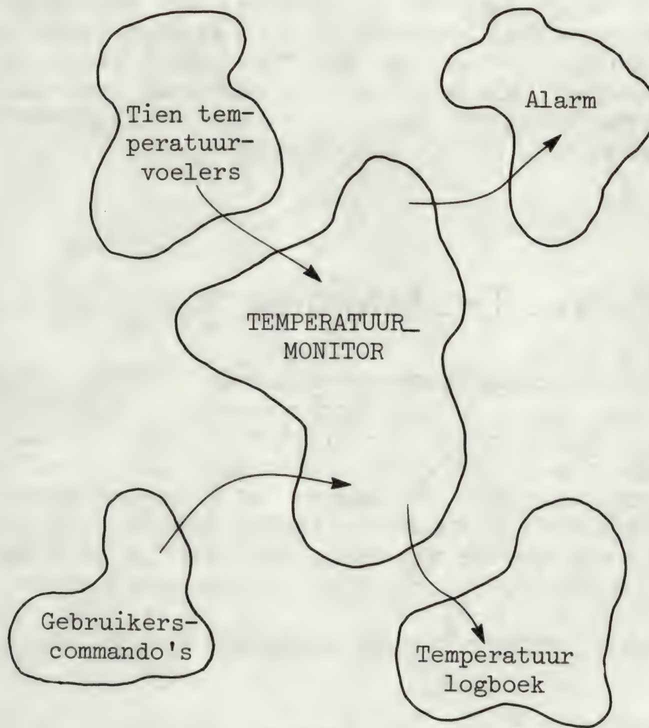
- parallelle verwerking
- real-time besturing
- exception handling
- speciale input/output besturing

We zeiden het al eerder: talen als FORTRAN, JOVIAL en Pascal beschikken niet over constructies om direct aan deze eisen te kunnen voldoen. De programmeur moet dan zijn toevlucht nemen tot faciliteiten van het onderliggende operating system en tot assembler procedures. De uiteindelijke programma's zijn dan slecht overdraagbaar en grote problemen zijn zo lastig oplosbaar. Ada werd speciaal ontworpen voor 'embedded system' omgevingen en met Ada's gereedschapsset voor het uitvoeren van parallelle taken en de behandeling van excepties tot onze beschikking, gaan we ons nu werpen op een multitasking embedded system probleem.



18.1 Definieer Het Probleem

Administratieve systemen zijn meestal sterk bepaald door processen voor het beheersen van invoer en uitvoer van gegevens; bij wetenschappelijke problemen ligt de nadruk meestal op gegevenstransformaties en staat dus de processor centraal. Bij ingebedde systemen is meestal nog iets anders het belangrijkste, namelijk het besturen en beheersen van zich in de werkelijke kloktijd afspelende processen of real-time processen. In dit soort systemen zijn bepaalde tijdsfactoren meestal de belangrijkste randvoorwaarden en moeten mogelijke exceptionele situaties correct worden opgevangen. In dit hoofdstuk behandelen we een bewakingssysteem, dat voortdurend een



Figuur 18-1 Het omgevingsbewakingsprobleem.

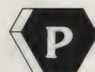
bepaalde omgeving controleert. We gebruiken natuurlijk weer onze objectgerichte ontwerpmethode om een oplossing in Ada te ontwikkelen.

Figuur 18-1 illustreert het probleemgebied. Volgens het plaatje gaat het hier om het in de gaten houden van temperaturen, maar dezelfde oplossing kan worden toegepast op tal van andere procesbesturingssystemen, die zaken controleren als druk, elektrische spanning of vloeistofniveaus, om maar een paar voorbeelden te noemen.

Binnen dit probleem bestaat onze wereld uit een aantal temperatuurvoelers. De gebruiker kan een bepaalde voeler in- of uitschakelen, kan grenswaarden instellen en de status van een voeler controleren. Verder is er een automatisch logboek, waarin regelmatig de status van alle voelers wordt genoteerd. Als één van de temperatuurgrenswaarden wordt bereikt, reageert het systeem met een alarmsignaal. Verder is het systeem in staat, incorrect functioneren te detecteren: als een sensor niet goed werkt zal het systeem eveneens een alarm geven. Een eenvoudige uitbreiding zou zijn om het systeem ook nog foutherstellend te maken door bijvoorbeeld een reservevoeler in te zetten, maar zo ver zullen we hier niet gaan.

We hebben nu een idee van het probleem als totaal. Het systeem vereist (en dit is typisch voor een ingebed systeem) communicatie met speciale randapparatuur voor input en output, zoals alarmbellen, logboek en sensoren. Als volgende stap zullen we een informele oplossingsstrategie formuleren en geven we ook de tijdslimieten, waarbinnen het systeem zal moeten functioneren.

18.2 Ontwikkel Een Informele Strategie



We hebben nog niet gesproken over de manier waarop de door de voelers gemeten waarden zullen worden voorgesteld en ook niet over de wijze waarop het alarm zal worden gegeven. Het zou ook te vroeg zijn als we dit nu al zouden doen en we zullen deze specificaties niet eens vermelden, maar daarmee wachten tot we op een lager abstractieniveau zijn afgedaald. De beslissing om het invullen van deze details uit te stellen heeft nog een voordeel: vaak weet de gebruiker of opdrachtgever dit soort gegevens in dit stadium niet eens precies te formuleren.

De informele strategie kunnen we nu bijvoorbeeld als volgt formuleren:

> Tien onderling onafhankelijke voelers bewaken voortdurend de temperatuur. In de beginsituatie zijn alle voelers uitgeschakeld. Elke voeler kan door het geven van een opdracht in- of uitgeschakeld worden en de status kan geregisteerd worden. Verder kan de onderste en bovenste limietwaarde van elke voeler worden ingesteld en als één van de ingeschakelde voelers een waarde meet buiten zijn limietwaarden, dan wordt onmiddellijk een alarm gegeven. De status van alle voelers moet om de vijftien minuten worden opgevraagd en geregisteerd (dit gebeurt via een hardware klok-interrupt). Indien een voeler niet binnen vijf seconden antwoord geeft, dan wordt verondersteld dat deze defect is en wordt opnieuw een alarmsignaal gegeven. Asynchroon kan een gebruikerscommando binnenkomen teneinde een bepaalde voeler in of uit te schakelen, om de limietwaarden in te stellen, of om de status van een bepaalde voeler direct te registreren. Nooit mag het buiten werking raken van het gebruikersinterface de bewaking van de ingeschakelde voelers verstoren.

18.3 Formaliseer De Strategie



De in het voorafgaande geformuleerde informele strategie gaan we nu, met gebruikmaking van Ada, formeler formuleren.

Identificeer de objecten en hun attributen

De objecten zijn te identificeren als de zelfstandige naamwoorden in de informele strategiebeschrijving, hun attributen worden beschreven door de bijbehorende bijvoeglijke naamwoorden. Beide categorieën onderstrepen we weer:

> Tien onderling onafhankelijke voelers bewaken voortdurend de temperatuur. In de beginsituatie zijn alle voelers uitgeschakeld. Elke voeler kan door het geven van een opdracht in- of uitgeschakeld worden en de status kan worden geregistreerd. Verder kan de onderste en de bovenste limietwaarde van elke voeler worden ingesteld en als één van de ingeschakelde voelers een waarde meet buiten zijn limietwaarden, dan wordt onmiddellijk een alarm gegeven. De status van alle voelers moet om de vijftien minuten worden opgevraagd en geregistreerd (dit gebeurt via een hardware klok-interrupt). Indien een voeler niet binnen vijf seconden antwoord geeft, dan wordt verondersteld dat deze defect is en wordt opnieuw een alarmsignaal gegeven. Asynchroon kan een gebruikerscommando binnenkomen teneinde een bepaalde voeler in of uit te schakelen, om de limietwaarden in te stellen, of om de status van een bepaalde voeler direct te registreren. Nooit mag het buiten werking raken van het gebruikersinterface de bewaking van de ingeschakelde voelers verstoren.

We onderscheiden in dit stadium van de oplossing de volgende objecten:

- ALARM
- VOELERS
- REGISTRATIE_APPARAAT
- TIJDKLOK
- GEBRUIKERSCOMMANDO

Weliswaar onderstreepten we nog andere zelfstandige naamwoorden, zoals limietwaarde, temperatuur en status, maar we zullen zien dat deze beter als attributen bij andere objecten kunnen worden beschouwd. Het object REGISTRATIE_APPARAAT komt niet expliciet in de beschrijving voor, maar het invoeren hiervan zal ons ontwerp leesbaarder en begrijpelijker blijken te maken.

De attributen van onze objecten zijn:

- ALARM
  signaal voor meting buiten limietwaarden en voor defecte voeler.
- VOELERS
  Tien voelers. Een voeler kan in- of uitgeschakeld zijn.
  Elke voeler heeft zijn eigen onderste en bovenste limietwaarde.
  De waarde van een voeler heeft alleen betekenis als deze ingeschakeld is.
  We kunnen de status en waarde van een voeler opvragen.
  Als een voeler geen antwoord geeft, dan kan dit worden gesignaleerd.

- REGISTRATIE_APPARAAT
  Status (toestand en waarde) van de voelers wordt geregistreerd.
- TIJDKLOK
  Geeft elke 15 minuten een interrupt signaal.
- GEBRUIKERSCOMMANDO
  Kan zijn: inschakelen, uitschakelen, stel limieten in en lees/registreer status.

Bepaal de bewerkingen op de objecten

De werkwoorden in onze beschrijving komen overeen met de bewerkingen, die op de objecten moeten kunnen worden uitgevoerd. Randvoorwaarden wat betreft reactietijden zullen we verder ook moeten noteren. We herhalen de tekst weer:

> Tien onderling onafhankelijke voelers bewaken voortdurend de temperatuur. In de beginsituatie zijn alle voelers uitgeschakeld. Elke voeler kan door het geven van een opdracht in- of uitgeschakeld worden en de status kan worden geregistreerd. Verder kan de onderste en bovenste limietwaarde van elke voeler worden ingesteld en als één van de ingeschakelde voelers een waarde meet buiten zijn limietwaarden, dan wordt onmiddellijk een alarm gegeven. De status van alle voelers moet om de vijftien minuten worden opgevraagd en geregistreerd (die gebeurt via een hardware klok-interrupt). Indien een voeler niet binnen vijf seconden antwoord geeft, dan wordt verondersteld dat deze defect is en wordt opnieuw een alarmsignaal gegeven. Asynchroon kan een gebruikerscommando binnenkomen teneinde een bepaalde voeler in of uit te schakelen, om de limietwaarden in te stellen, of om de status van een bepaalde voeler direct te registreren. Nooit mag het buiten werking raken van het gebruikersinterface de bewaking van de ingeschakelde voelers verstoren.

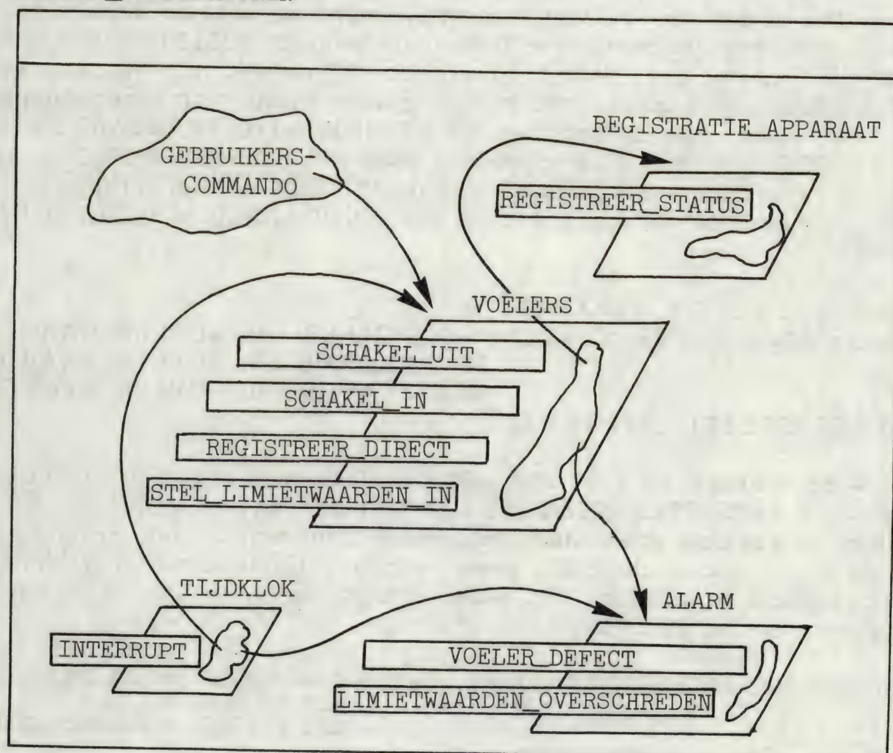
We kunnen nu de volgende bewerkingen onderscheiden:

- ALARM
  - VOELER_DEFECT
  - LIMIETWAARDEN_OVERSCHREDEN
- VOELERS
  - SCHAKEL_UIT
  - SCHAKEL_IN
  - REGISTREER_DIRECT
  - STEL_LIMIETWAARDEN_IN
- REGISTRATIE_APPARAAT
  - REGISTREER_STATUS
- TIJDKLOK
  - INTERRUPT
- GEBRUIKERSCOMMANDO
  - HAAL_BINNEN

Omschrijvingen als *voortdurend*, *onderling onafhankelijk* en *asynchroon* maken duidelijk dat de objecten als tegelijkertijd actief kunnen worden beschouwd.

Stel de interfaces vast

Hoe hangen onze objecten onderling samen en hoe communiceren zij? In figuur 18-2 hebben we het ontwerp schematisch weergegeven. We kozen hier een enigszins andere structuur dan bij de vorige oplossingen. Wel is ook hier de hoofdstructuur een subprogramma, maar in dit geval namen we al onze objecten binnen deze hoofdeenheid op. In dit geval hebben we immers maar één verschijningsvorm (kopie) van elk object nodig, dus hoeft geen der objecten als een bibliotheekeenheid te worden geïmplementeerd. Ook zagen we al dat alle objecten zullen optreden als onderling onafhankelijke grootheden en we kiezen daarom hier voor een aantal parallelle processen of taken; dit is de meest realistische afspiegeling van ons in de werkelijkheid waargenomen probleemgebied. Indien nodig kunnen we van onze objecten later nog altijd bibliotheekeenheden maken. Hoewel een taak geen compileerbare eenheid is, zagen we in hoofdstuk 16 dat een taak in een pakket kan worden opgenomen en zo dus toch als bibliotheekeenheid kan functioneren.



Figuur 18-2 **Ontwerp voor** BEWAAK_TEMPERATUREN.

Uit de figuur blijkt dat de verbindingen tussen de grootheden eenvoudig zijn en gering in aantal. Zowel GEBRUIKERSCOMMANDO als TIJDKLOK communiceren met VOELERS. VOELERS op hun beurt sturen berichten naar ALARM en naar REGISTRATIE_APPARAAT. Door deze relaties is meteen vastgelegd hoe de eenheden ten opzichte van elkaar zichtbaar moeten zijn. De voor de oplossing benodigde interfaces kunnen formeel worden beschreven in de taakspecificaties bij de objecten. De nadere uitwerking hiervan zullen we echter nog even uitstellen.

We gaan onze oplossing 'top-down', dus beginnend op het hoogste abstractieniveau, ontwikkelen. In Ada betekent dat allereerst het uitschrijven van de specificaties van de programma-eenheden en vervolgens de uitwerking van de bijbehorende bodies. In de specificaties van ALARM staat, dat het in het geval van LIMIETWAARDEN_OVERSCHREDEN mogelijk moet zijn aan te geven welke voeler het alarm veroorzaakte. In Ada gaat de ALARM specificatie er dan bijvoorbeeld zo uitzien:

```
task ALARM is
  entry VOELER_DEFECT;
  entry LIMIETWAARDEN_OVERSCHREDEN(DOOR_VOELER : in VOELER_NAAM);
end ALARM
```

Het gebruikte type VOELER_NAAM hebben we nog niet gedefinieerd. Dat is ook niet zo verwonderlijk, want we hebben immers nog geen beslissing genomen over de manier waarop VOELER moet worden voorgesteld. Het gaat daar trouwens op dit moment nog niet om; we zijn nu louter geïnteresseerd in de logische relaties en eigenschappen.

Vervolgens specificeren we het REGISTRATIE_APPARAAT. Als we REGISTREER_STATUS uitvoeren voor een bepaalde VOELER, dan gaat het niet alleen om de naam van de VOELER en zijn huidige waarde, maar ook om zijn toestand (INGESCHAKELD of UITGESCHAKELD):

```
task REGISTRATIE_APPARAAT is
  entry REGISTREER_STATUS(VAN_VOELER : in VOELER_NAAM;
                          MET_WAARDE : in VOELER_WAARDE;
                          MET_STATUS : in VOELER_STATUS);
end REGISTRATIE_APPARAAT;
```

Weer voerden we twee nog niet gedefinieerde typen in: VOELER_WAARDE en VOELER_STATUS. Ook van deze typen zullen we de nadere uitwerking even uitstellen. Merk trouwens op hoe ervoor gezorgd is het geheel door het geven van begrijpelijke namen zo leesbaar mogelijk te houden. Een aanroep naar de entry kan er nu bijvoorbeeld als volgt uitzien:

```
REGISTRATIE_APPARAAT.REGISTREER_STATUS(VAN_VOELER => BROEIKAS,
                                       MET_WAARDE => 1.73,
                                       MET_STATUS => INGESCHAKELD);
```

TIJDKLOK kan er al heel eenvoudig uitzien; in de informele strategie werd gesproken van een hardware interrupt en we veronderstellen, dat dit interrupt een sprong naar geheugenadres 8E (hexadecimaal) veroorzaakt en dat daar een service routine begint:

```
task TIJDKLOK is
  entry INTERRUPT;
  for INTERRUPT use at 16#8E#;
                          -- pakket SYSTEM moet zichtbaar zijn
end TIJDKLOK;
```

TIJDKLOK zorgt dat op vaste tijdstippen direct via REGISTREER_DIRECT de statusgegevens van de voelers in VOELERS worden vastgelegd.

VOELERS kan er dan als volgt uitzien:

```
task VOELERS is
  entry SCHAKEL_UIT              (VOELER      : in VOELER_NAAM);
  entry SCHAKEL_IN               (VOELER      : in VOELER_NAAM);
  entry REGISTREER_DIRECT        (VAN_VOELER  : in VOELER_NAAM);
  entry STEL_LIMIETWAARDEN_IN(VOOR_VOELER : in VOELER_NAAM;
                              ONDERGRENS  : in VOELER_WAARDE;
                              BOVENGRENS  : in VOELER_WAARDE);
end VOELERS;
```

Alleen GEBRUIKERSCOMMANDO moeten we nu nog definiëren; we kiezen hier niet voor een pakket, maar voor een enumeratietype:

```
type COMMANDO is (SCHAKEL_IN,SCHAKEL_UIT,
                  REGISTREER_STATUS,
                  STEL_LIMIETWAARDEN_IN).
GEBRUIKERSCOMMANDO : COMMANDO;
```

Het ophalen van een commando kan gebeuren met de in Ada's standaardpakket voor I/O faciliteiten opgenomen GET operatie.

Nu alle eenheden en alle interfaces met de eenheden zijn beschreven, is de volgende stap het uitwerken van de operaties op de objecten.

Programmeer de bewerkingen

We beginnen met de uitwerking van het hoofdprogramma, BEWAAK_TEMPERATUREN. Zoals meestal in multitasking situaties is het voornaamste doel van het hoofdprogramma, de taken te besturen en te activeren of te beëindigen. In Ada is een expliciete activeringsinstructie trouwens niet noodzakelijk: elke taak wordt onmiddellijk bij de uitwerking van de programmatekst actief. (Zie nog eens hoofdstuk 16.) Verder gaat het hier om een niet aflopend proces (dat komt vaak voor bij ingebedde systemen), dus over beëindigen van taken hoeven we ons hier ook al niet druk te maken.

In Ada kan het hoofdprogramma worden behandeld als hoofdtaak, waarvan alle overige taken afhankelijk zijn. De top-down methode toepassend kan een eerste versie van ons hoofdprogramma er zo uitzien:

```
procedure BEWAAK_TEMPERATUREN is
   -- lokale typedeclaraties
   -- ALARM taakspecificatie
   -- REGISTRATIE_APPARAAT taakspecificatie
   -- VOELERS taakspecificatie
   -- TIJDKLOK taakspecificatie
   -- GEBRUIKERSCOMMANDO declaratie
   -- taakbodies
begin
   -- manipulatie van GEBRUIKERSCOMMANDO
end BEWAAK_TEMPERATUREN;
```

De volgorde van de declaraties komt overeen met de schematische voorstelling in figuur 18-2. Omdat Ada's regels betreffende zichtbaarheid van grootheden eisen, dat een grootheid gedeclareerd moet zijn voordat ernaar verwezen kan worden, worden de taakbodies bij elkaar geplaatst aan het einde van het declaratiegedeelte van het hoofdprogramma. Iedere taak kan op die manier de specificatie van alle andere 'zien'. De taakbodies zelf zullen we als subeenheden declareren. Dat wil zeggen: we noemen ze hier slechts, maar werken ze afzonderlijk uit en compileren ze ook afzonderlijk. Dat houdt de hoofdprogrammatekst overzichtelijk en past uitstekend bij de top down ontwerpmethode (hierop zullen we in hoofdstuk 20 nog nader ingaan).

Het moment is nu aangebroken om nader beslissingen te nemen over de voorstellingswijze van de tot nu toe nog niet uitgewerkte typen VOELER_NAAM, VOELER_STATUS en VOELER_WAARDE. Voor VOELER_NAAM zouden we een nummering met de waarden 1 tot en met 10 kunnen kiezen, maar erg beeldend is dat niet. Duidelijker is een enumeratietype met expliciete namen voor de sensoren, die bijvoorbeeld informatie over hun lokatie geven:

```
type VOELER_NAAM is (HAL,KANTOOR,MAGAZIJN,
                     OPSLAGRUIMTE,TERMINALKAMER,
                     BIBLIOTHEEK,COMPUTERRUIMTE,
                     ONTVANGSTRUIMTE,LAADPLATFORM,
                     BEZEMKAST);
```

Een voeler kan UITGESCHAKELD of INGESCHAKELD zijn:

```
type VOELER_STATUS is (UITGESCHAKELD,INGESCHAKELD);
```

Aan VOELER_WAARDE voegen we een stapgrootte toe om de precisie te definiëren:

```
type VOELER_WAARDE is delta 0.5 range 0.0 .. 100.0;
```

Het is vervolgens tijd om te denken aan de mogelijkheden voor interactie met de gebruiker: waarden van de typen VOELER_NAAM, VOELER_WAARDE en COMMANDO moeten kunnen worden ingelezen en afgedrukt en de objecten ONDERGRENS, BOVENGRENS, NAAM en WAARDE zullen we toevoegen om de communicatie te vereenvoudigen. Voor invoer en uitvoer zullen we gebruik maken van het standaardpakket TEXT_IO en het eenvoudigst gaat dat als we voor elk type, dat gelezen of geschreven moet worden, een generiek pakket creëren:

```
package COMMANDO_IO is new ENUMERATION_IO(COMMANDO);
use COMMANDO_IO;
package VOELER_NAAM_IO is new ENUMERATION_IO(VOELER_NAAM);
use VOELER_NAAM_IO;
package VOELER_WAARDE_IO is new FIXED_IO(VOELER_WAARDE);
use VOELER_WAARDE_IO;
```

Deze pakketcreaties (de Ada term luidt: 'instantiaties') worden aan het begin van het declaratiedeel van het hoofdprogramma geplaatst en dit kan er dus zo uitzien (er is in ruime mate van spatiëring gebruik gemaakt om het geheel zo leesbaar mogelijk te maken):

```
with TEXT_IO,SYSTEM;
use TEXT_IO;
procedure BEWAAK_TEMPERATUREN is
  --
  type COMMANDO      is (SCHAKEL_UIT,          SCHAKEL_IN,
                         REGISTREER_STATUS,STEL_LIMIETWAARDEN_IN);
  --
  type VOELER_NAAM is (HAL,               KANTOOR,          MAGAZIJN,
                         OPSLAGRUIMTE,  TERMINALKAMER   BIBLIOTHEEK,
                         COMPUTERRUIMTE,ONTVANGSTRUIMTE,LAADPLATFORM,
                         BEZEMKAST);
  type VOELER_STATUS is (UITGESCHAKELD,INGESCHAKELD);
  type VOELER_WAARDE is delta 0.5 range 0.0 .. 100.0;
  --
  package COMMANDO_IO        is new ENUMERATION_IO(COMMANDO);
  use     COMMANDO_IO;
  package VOELER_NAAM_IO     is new ENUMERATION_IO(VOELER_NAAM);
  use     VOELER_NAAM_IO;
  package VOELER_WAARDE_IO is new FIXED_IO(VOELER_WAARDE);
  use     VOELER_WAARDE_IO;
  --
  task ALARM is
    entry VOELER_DEFECT;
    entry LIMIETWAARDEN_OVERSCHREDEN(DOOR_VOELER : in VOELER_NAAM);
  end ALARM;
  --
  task REGISTRATIE_APPARAAT is
    entry REGISTREER_(STATUS(VAN_VOELER : in VOELER_NAAM;
                              MET_WAARDE : in VOELER_WAARDE;
                              MET_STATUS : in VOELER_STATUS);
  end REGISTRATIE_APPARAAT;
```

```
task TIJDKLOK is
  entry INTERRUPT;
  for INTERRUPT use at 16#8E#;
end TIJDKLOK;
--
BOVENGRENS           : VOELER_WAARDE;
ONDERGRENS           : VOELER_WAARDE;
NAAM                 : VOELER_NAAM;
GEBRUIKERSCOMMANDO   : COMMANDO;
WAARDE               : VOELER_WAARDE;
--
task body ALARM                   is separate;
task body REGISTRATIE_APPARAAT    is separate;
task body VOELERS                 is separate;
task body TIJDKLOK                is separate;
```

Nu volgt de body van het hoofdprogramma. Als we onze informele strategiebeschrijving nog eens doorlezen, dan blijkt daaruit dat commando's van de gebruiker kunnen worden opgevraagd en dat vervolgens de bijbehorende bewerking uit VOELERS moet worden uitgevoerd. Het opvragen doen we met een aanroep van de procedure GET uit het standaardpakket TEXT_IO. In de specificaties staat vermeld dat eventuele fouten in de communicatie met de gebruiker nooit de werking van het systeem mogen verstoren. We zorgen er dus voor dat dit deel van de programmatuur alle fouten kan opvangen door middel van een exception handling procedure, die reageert op DATA_ERRORs. We gaan er voorlopig van uit, dat het pakket TEXT_IO de exceptie DATA_ERROR genereert als er niet toegelaten input van de gebruiker komt (bijvoorbeeld SCHAKEL_IUT in plaats van SCHAKEL_UIT).

```
begin
  loop
    begin   -- hier begint een lokaal blok met exception handler
      PUT("Voer commando in: ");
      GET(GEBRUIKERSCOMMANDO);
      NEW_LINE;
      PUT_LINE("Commando geaccepteerd");
      case GEBRUIKERSCOMMANDO is
        when SCHAKEL_UIT =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          VOELERS.SCHAKEL_UIT(VOELER =>NAAM);
          PUT_LINE("Voeler uitgeschakeld");
        when SCHAKEL_IN =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          VOELERS.SCHAKEL_IN(VOELER =>NAAM);
          PUT_LINE("Voeler ingeschakeld");
        when REGISTREER_STATUS =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          VOELERS.REGISTREER_DIRECT(VAN_VOELER => NAAM);
          PUT_LINE("Voeler status geregistreerd");
```

```
        when STEL_LIMIETWAARDEN_IN =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          PUT("Geef ondergrens: ");
          GET(ONDERGRENS);
          NEW_LINE;
          PUT("Ondergrens geaccepteerd");
          PUT ("Geef bovengrens: ");
          GET(BOVENGRENS);
          NEW_LINE;
          PUT ("Bovengrens geaccepteerd");
          VOELERS.STEL_LIMIETWAARDEN_IN(VOOR_VOELER => NAAM,
                                        ONDERLIMIET => ONDERGRENS,
                                        BOVENLIMIET => BOVENGRENS);
          PUT_LINE("Limietwaarden ingesteld);
      end case;
      exception
        when DATA_ERROR =>
          PUT_LINE("Invoer niet correct, opnieuw alstublieft");
      end;
  end loop;
end BEWAAK_TEMPERATUREN;
```

Invoer van de gebruiker wordt op een standaardmanier geaccepteerd en als de gebruiker onjuiste waarden intikt ontstaat een exceptie, die door de lokale exception handler wordt behandeld. Natuurlijk zou de communicatie nog heel wat gebruikersvriendelijker kunnen plaatsvinden; deze 'human engineering aspecten', die op zichzelf heel belangrijk zijn, laten wij ter nadere uitwerking aan de lezer over.

Nu kunnen we de bodies uitwerken van de pakketten, die in het hoofdprogramma worden gedeclareerd. In dit hoofdstuk werken we de eenheden afzonderlijk uit; in Appendix F wordt het programma in zijn geheel nog eens afgedrukt.

Volgens de informele strategie kan elke sensor een alarm geven en is het ook nog mogelijk dat we tegelijkertijd ontdekken dat één of meer sensoren niet meer reageren. In een dergelijke situatie moeten veel berichten worden verstuurd en daarom maken we van ALARM een taak. We kunnen er dan zeker van zijn dat de entry aanroepen LIMIETWAARDE_OVERSCHREDEN en VOELER_DEFECT in een wachtrij komen en uiteindelijk verwerkt zullen worden. Nu we hier zijn aangeland specificeren we de alarmtoestand nog wat nader: als er een alarm optreedt dat moet er een waarschuwingslampje gaan branden, dat door de gebruiker moet worden uitgeschakeld. Verder zullen we uitgaan van vaste geheugenadressen voor IO-poorten:

```
16#0010#                     -- adres voor 'voeler defect'
16#0011# tot en met 16#001A# -- adressen voor 'limietwaarden
                             -- overschreden'
```

Om het waarschuwingslampje te laten branden moeten alle 16 bits op zo'n adres op één gezet worden (16#FFFF#).

Na deze nadere specificatie wordt dit de body van de taak ALARM:

```
with SYSTEM;
separate (BEWAAK_TEMPERATUREN)
task body ALARM is
--
  BITS    : constant := 1;
  WOORDEN : constant := 16 * BITS;
  --
  type LAMP      is (UIT,AAN);
  for  LAMP'SIZE use 1 * WOORDEN;
  for  LAMP      use (UIT =>16#0000#,AAN => 16#FFFF#);
  DEFECT_LAMP : LAMP := UIT;
  for  DEFECT_LAMP use at 16#0010#;
--
type GRENSWAARDE_CONTROLE is array (VOELER_NAAM) of LAMP;
for  GRENSWAARDE_CONTROLE use (VOELER_NAAM'POS(VOELER_NAAM'LAST)
                              + 1) * WOORDEN;
GRENSWAARDE_LAMP : GRENSWAARDE_CONTROLE
                   := GRENSWAARDE_CONTROLE'(others => UIT);
for  GRENSWAARDE_LAMP use at 16#0011#;
--
begin
  loop
    select
      accept VOELER_DEFECT do
        DEFECT_LAMP := AAN;
      end VOELER_DEFECT;
    or
      accept LIMIETWAARDEN_OVERSCHREDEN(OP_VOELER : in VOELER_NAAM) do
        GRENSWAARDE_LAMP(OP_VOELER) := AAN;
      end LIMIETWAARDEN_OVERSCHREDEN;
    end select;
  end loop;
end ALARM;
```

In de **separate**-clausule wordt de 'ouder' van de subeenheid ALARM genoemd. De werking van ALARM is vrij eenvoudig: zodra de body wordt uitgewerkt, wordt de taak geactiveerd en wanneer een fout wordt gedetecteerd wordt via **select** de bijbehorende actie geselecteerd en gaat het desbetreffende lampje branden.

De bitpatronen voor UIT en AAN worden met behulp van een representatiespecificatie aangegeven. DEFECT_LAMP beslaat één woord in het geheugen; GRENSWAARDE_LAMP beslaat tien opeenvolgende woorden.

Bij de specificatie van het SIZE-attribuut hadden we direct 10 * WOORDEN kunnen gebruiken, in plaats daarvan maakten we gebruik van het aantal mogelijke waarden van VOELER_NAAM. Zou het aantal voelers gewijzigd worden, dan wordt automatisch de overeenkomstige hoeveelheid geheugen voor het array van LAMP elementen gereserveerd en dit bevordert de onderhoudbaarheid.

Voor de uitwerking van de body van REGISTRATIE_APPARAAT veronderstellen we, dat we over een pakket APPARAAT_IO beschikken, waarin de communicatie met de registratie-apparatuur geregeld

wordt. Met behulp van de procedure PUT kunnen dan waarden worden geschreven. Dit is in het algemeen een geschikte werkwijze: in plaats van ons bezig te moeten houden met technische details op het gebied van input en output van gegevens, beschouwen we APPARAAT_IO als een 'black box', waarbinnen deze details worden opgelost.

De taakbody van REGISTRATIE_APPARAAT heeft een eenvoudige structuur: een eindeloze wachtlus op entry-aanroepen van REGISTREER_STATUS:

```
with APPARAAT_IO;
separate (BEWAAK_TEMPERATUREN)
task body REGISTRATIE_APPARAAT is
  begin
    loop
      accept REGISTREER_STATUS(VAN_VOELER : in VOELER_NAAM;
                               MET_VOELER : in VOELER_WAARDE;
                               MET_STATUS : in VOELER_STATUS) do
        APPARAAT_IO.PUT(VAN_VOELER);
        APPARAAT_IO.PUT(MET_WAARDE);
        APPARAAT_IO.PUT(MET_STATUS);
      end REGISTREER_STATUS;
    end loop;
  end REGISTRATIE_APPARAAT;
```

De REGISTRATIE_APPARAAT body zou nog verbeterd kunnen worden als de statusinformatie met behulp van een buffer wordt verwerkt, zodat taken die een entry REGISTREER_STATUS aanroepen niet hoeven te wachten totdat het registratie-apparaat klaar is met de fysieke verwerking. Deze verbetering wordt als oefening aan de lezer overgelaten. (Het generieke FIFO pakket uit hoofdstuk 14 bevat alle benodigde hulpmiddelen.)

Vervolgens werken we de taak VOELERS uit. Volgens de specificaties zijn er tien voelers. Hun waarden worden ingelezen vanaf IO-poorten met vaste geheugenadressen, beginnend bij adres 16#0100#. Er wordt een geheeltallige waarde ingelezen, die de temperatuur weergeeft met een precisie van een halve graad. Die waarde moet vervolgens met 0.5 worden vermenigvuldigd om hem bij het type VOELER_WAARDE te laten passen. In de uitwerking zullen we laten zien hoe dit met behulp van representatiespecificaties kan worden gerealiseerd.

De taak wordt ook weer gestructureerd als een eeuwige lus, waarin op entry-aanroepen wordt gewacht. Als er geen aanroep meer moet worden verwerkt, dan worden de waarden van de ingeschakelde voelers ingelezen (als onderdeel van de **else**-clausule). Entries SCHAKEL_UIT, SCHAKEL_IN, REGISTREER_STATUS of STEL_LIMIET-WAARDEN_IN zullen via een alternatieve **select**-instructie (zie hoofdstuk 16) worden verwerkt.

Binnen de taak definiëren we een VOELER_RECORD met de componenten ONDERGRENS, BOVENGRENS en WAARDE; een structuur die op natuurlijke wijze voortkomt uit onze informele strategie. Het

SET_PAKKET uit hoofdstuk 15 gebruiken we om de toestanden van de voelers bij te houden; een voeler is ingeschakeld als deze een element is van de verzameling ACTIEVE_VOELERS.

```
with SET_PAKKET, SYSTEM;
separate (BEWAAK_TEMPERATUREN)
task body VOELERS is
  --
  BITS    : constant := 1;
  WOORDEN : constant := 16 * BITS;
  --
  type VOELER_RECORD is record
    BOVENGRENS : VOELER_WAARDE := VOELER_WAARDE'LAST;
    ONDERGRENS : VOELER_WAARDE := VOELER_WAARDE'FIRST;
    WAARDE     : VOELER_WAARDE := VOELER_WAARDE'FIRST;
  end record;
  type VOELER_GROEP is array (VOELER_NAAM) of VOELER_RECORD;
  VOELER         : VOELER_GROEP;
  --
  package VOELER_SET is new SET_PAKKET(UNIVERSUM => VOELER_NAAM);
  use VOELER_SET;
  ACTIEVE_VOELERS : SET := NULL_SET;
  --
  type VOELER_POORT is range 0 .. (2 ** WOORDEN - 1);
  for VOELER_POORT'SIZE use 1 * WOORDEN;
  type VOELER_LIJST is array (VOELER_NAAM) of VOELER_POORT;
  for VOELER_LIJST'SIZE use (VOELER_NAAM'POS(VOELER_NAAM'LAST)
                            + 1) * WOORDEN;
  VOELER_MAP : VOELER_LIJST;
  for VOELER_MAP use at 16#0100#;
  --
begin
  loop
    select
      accept SCHAKEL_UIT(VOELER : in VOELER_NAAM) do
        ACTIEVE_VOELERS := ACTIEVE_VOELERS - VOELER;
      end SCHAKEL_UIT;
    or
      accept SCHAKEL_IN(VOELER : in VOELER_NAAM) do
        ACTIEVE_VOELERS := ACTIEVE_VOELERS + VOELER;
      end SCHAKEL_IN;
    or
      accept REGISTREER_DIRECT(VAN_VOELER : in VOELER_NAAM) do
        if IS_ELEMENT(VAN_VOELER,VAN_SET => ACTIEVE_VOELERS) then
          REGISTRATIE_APPARAAT.REGISTREER_STATUS
                           (VAN_VOELER,
                            VOELER(VAN_VOELER).WAARDE,
                            MET_STATUS => INGESCHAKELD);
        else
          REGISTRATIE_APPARAAT.REGISTREER_STATUS
                           (VAN_VOELER,
                            MET_WAARDE => VOELER_WAARDE'FIRST,
                            MET_STATUS => UITGESCHAKELD);
        end if;
      end REGISTREER_DIRECT;
```

```
  or
    accept STEL_LIMIETWAARDEN_IN(VOOR_VOELER : in  VOELER_NAAM;
                                 ONDERGRENS  : in  VOELER_WAARDE;
                                 BOVENGRENS  : in  VOELER_WAARDE) do
      VOELER(VOOR_VOELER).ONDERGRENS := ONDERGRENS;
      VOELER(VOOR_VOELER).BOVENGRENS := BOVENGRENS;
    end STEL_LIMIETWAARDEN_IN;
  else
    for I in VOELER_NAAM
      loop
        if IS_ELEMENT(I,VAN_SET => ACTIEVE_VOELERS) then
          VOELER(I).WAARDE := (VOELER_MAP(I) * VOELER_WAARDE(0.5));
          if (VOELER(I).WAARDE < VOELER(I).ONDERGRENS) or
            (VOELER(I).WAARDE > VOELER(I).BOVENGRENS) then
           ALARM.LIMIETWAARDEN_OVERSCHREDEN(I);
          end if;
        end if;
      end loop;
    end select;
  end loop;
end VOELERS;
```

De geheeltallige VOELER_MAP waarden worden als volgt naar waarden van het type VOELER_WAARDE (fixed-point waarden) geconverteerd: eerst wordt de VOELER_MAP waarde met 0.5 vermenigvuldigd. Een vermenigvuldiging van een geheeltallige waarde met een fixed-point waarde heeft automatisch een fixed-point waarde als resultaat, dus mag deze waarde direct aan VOELER(I).WAARDE worden toegekend. Het resultaat ziet er misschien wat omslachtig uit, maar in ieder geval hebben we volledige controle over de typeconversie - voor verrassingen komen we in Ada wat dit betreft niet te staan.

Nu rest ons nog de taak TIJDKLOK. We veronderstellen dat er in de hardware een interne klok aanwezig is, die elke minuut voor een hardware interrupt zorgt. Dit interrupt heeft een sprong van de verwerking tot gevolg naar de instructie op geheugenadres 16#004#. Er worden vijftien tikken van de klok geteld (de specificaties spreken immers van een registratie om de vijftien minuten) en dan wordt de entry REGISTREER_DIRECT aangeroepen voor alle mogelijke waarden van VOELER_NAAM. Als niet binnen vijf seconden een antwoord van de voeler wordt terug ontvangen, dan wordt de entry VOELER_DEFECT aangeroepen. De taakbody kan er nu zo uitzien:

```
separate (BEWAAK_TEMPERATUREN)
task body TIJDKLOK is
  MINUTEN : constant := 1;
  type INTERVAL is range 0 .. 15;
  TIKKEN : INTERVAL := 0;
```

```
begin
  loop
    accept INTERRUPT do
      TIKKEN := TIKKEN + 1;
      if TIKKEN = 15 * MINUTEN then
        for I in VOELER_NAAM
          loop
            select
              VOELERS.REGISTREER_DIRECT(VAN_VOELER => I);
            or
              delay 5.0;
              ALARM.VOELER_DEFECT;
            end select;
          end loop;
        TIKKEN := 0;
      end if;
    end INTERRUPT;
  end loop;
end TIJDKLOK;
```

MINUTEN werd gebruikt terwille van de leesbaarheid: 15 *MINUTEN is duidelijker dan alleen maar 15. Een goede compiler moet zelf ontdekken dat 15 * MINUTEN invariant is en dus maar éénmaal berekend hoeft te worden. In dat geval leidt de extra vermenigvuldiging niet tot een verlenging van de verwerkingstijd.

Hiermee zijn we klaar met de uitwerking van BEWAAK_TEMPERATUREN. De uiteindelijke oplossing lijkt misschien nogal wat omhaal van woorden nodig te hebben. In Appendix F vatten we deze daarom nog eens samen. Wij hopen zo langzamerhand duidelijk te hebben gemaakt dat Ada heel geschikt is voor het beschrijven van betrekkelijk ingewikkelde oplossingen op een leesbare manier. Ada zorgt er verder voor dat deze oplossing betrouwbaar is (alle interfaces zijn precies beschreven) en onderhoudbaar (een wijziging in de tijdspecificaties zou bijvoorbeeld vrij eenvoudig zijn).

Oefeningen

1. Herschrijf de body van BEWAAK_TEMPERATUREN en zorg daarbij voor gebruikersvriendelijker reacties op fouten van de gebruiker bij invoer van gegevens.

2. Herschrijf de body van REGISTRATIE_APPARAAT en maak gebruik van een buffer voor de ingevoerde gegevens. Het in hoofdstuk 13 behandelde FIFO pakket kan u hierbij van pas komen. Aan dit pakket zou wellicht nog de functie IS_LEEG kunnen worden toegevoegd om de eindoplossing wat efficiënter te maken.

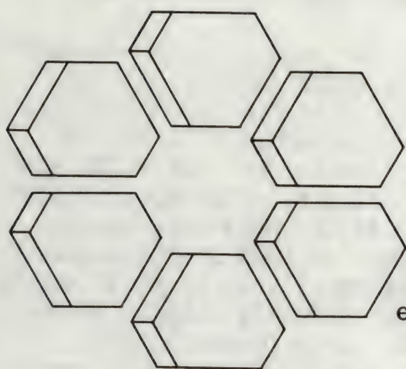
*3. In de taak VOELERS zou nog meer parallellisme kunnen worden ingebouwd als we een afzonderlijke taak voor elke voeler zouden creëren. Herschrijf de body, zodat deze tien taken als 'kinderen' bevat, maar zonder de specificatie van VOELERS te veranderen.

4. Veronderstel dat het systeem, dat we in dit hoofdstuk uitwerkten, op een 8-bits computer moet worden overgezet. Dit heeft tot gevolg, dat de componenten van VOELER_MAP nu elk twee geheugenwoorden beslaan. Wijzig het systeem in dit opzicht (neem daarbij aan, dat de adreslokaties dezelfde blijven).

*5. Breng een wijziging in het systeem aan, die erop neerkomt dat, als de temperatuur in COMPUTERRRUIMTE dichter dan tien graden bij de ondergrens of de bovengrens komt, de status van elke voeler niet meer elke vijftien, maar elke vijf minuten wordt geregistreerd.

Pakket 7

SYSTEEMONTWIKKELING



Een complex systeem dat werkt is vrijwel altijd ontwikkeld uit een eenvoudig systeem dat werkte.

John Gall
The Fifteenth Law of Systemantics [1]

19 INPUT/OUTPUT

Het zorgen voor faciliteiten voor het invoeren en weergeven van gegevens (Input/Output of kortweg I/O genaamd) is altijd zoiets geweest als naar de tandarts gaan: de ontwerpers van een programmeertaal weten dat het moet geburen, maar ze schijnen te hopen dat deze noodzaak, door het uit te stellen, vanzelf weer zal verdwijnen. Daarom zitten we nu met talen als ALGOL, waarin de I/O mogelijkheden per implementatie van de taal verschillen, en met Pascal, waarin ze zeer beperkt zijn. In Ada wordt I/O niet behandeld als de spreekwoordelijke nagel aan de doodskist; doordat Ada wegens het pakketmechanisme een uitbreidbare taal is, hoeven in de taal zelf helemaal geen speciale extra faciliteiten voor I/O te worden aangebracht.

In veel programmeertalen worden we juist opgescheept met veel meer I/O mogelijkheden dan we ooit nodig zullen hebben. Als we in FORTRAN bijvoorbeeld maar één bepaalde vorm van geformatteerde I/O gebruiken, dan slepen de meeste implementaties toch de hele enorme standaardbibliotheekeenheid mee, waarin voor werkelijk elke vorm van invoer en weergave een voorziening is, of we dit nu willen of niet. Vooral binnen het gebied van de ingebedde computersystemen is deze werkwijze onaanvaardbaar: veel te veel kostbaar geheugen wordt hierdoor onnodig in beslag genomen. Vaak is bij dit soort toepassingen ook geen sprake van de standaardhulpmiddelen voor het invoeren en weergeven van gegevens, zoals toetsenborden, beeldschermen en printers, maar van de een of andere 'black box' met zijn eigen I/O protocol.

We willen dus de mogelijkheid hebben, onze eigen I/O routines te ontwikkelen voor bepaalde gespecialiseerde apparatuur. Daarbij willen we ook kunnen beschikken over voorgedefinieerde eenheden voor I/O van de meest voorkomende datatypen, zoals lettertekens, gehele en reële getallen. In dit hoofdstuk zullen we laten zien, dat Ada in dit alles, en zelfs in meer, voorziet.

Er zijn in Ada voorgedefinieerde pakketten voor Input/Output op drie niveaus:

- INPUT_OUPUT — twee generieke pakketten voor elementen van elk type
  - SEQUENTIAL_IO — I/O voor sequentiële bestanden
  - DIRECT_IO — I/O direct toegankelijke (random access) bestanden

- TEXT_IO — een pakket voor het verwerken van tekst
- LOW_LEVEL_IO — een door de gebruiker te definiëren pakket voor primitieve I/O

Ook is er ook nog een pakket IO_EXCEPTIONS waarin de excepties worden gedefinieerd voor de pakketten SEQUENTIAL_IO, DIRECT_IO en TEXT_IO. Zie Appendix C voor nadere specificatie van de in Ada voorgedefinieerde I/O-pakketten.

In dit hoofdstuk worden alleen Ada's voorgedefinieerde I/O faciliteiten behandeld. Juist wegens de uitbreidbaarheid van de taal kunnen I/O faciliteiten naar believen worden toegevoegd en dit betekent, dat de werkwijze bij het verwerken van invoer en weergave van gegevens per lokatie enigszins kan verschillen. Toch heeft dit geen nadelige invloed op de overdraagbaarheid van programmatuur; ook gebruikergedefinieerde I/O-pakketten moeten immers steeds binnen de taal zelf en dus volgens de regels van de taal worden beschreven. Het gaat dus nooit om faciliteiten, die boven de taal uitgaan, maar altijd om uitbreidingen binnen Ada zelf.

19.1 Input/Output Van Numerieke Gegevens



De generieke pakketten SEQUENTIAL_IO en DIRECT_IO bevatten alle mogelijkheden voor het invoeren en weergeven van elementen van bepaalde typen. Deze pakketten zijn bedoeld voor binaire I/O: de gegevens worden voorgesteld door rijtjes bits en niet door voor mensen leesbare tekens (die natuurlijk ook weer binair gecodeerd worden). Beide pakketten bieden dezelfde faciliteiten; SEQUENTIAL_IO is bedoeld voor sequentiële bestanden en DIRECT_IO voor 'direct access' bestanden.

De volgende operaties zijn in beide generieke pakketten gedefinieerd:

- CLOSE
- CREATE
- DELETE
- OPEN
- READ
- RESET
- WRITE

En de volgende functies:

- END_OF_FILE
- FORM
- IS_OPEN
- MODE
- NAME

Verder kent het pakket DIRECT_IO nog de procedure SET_INDEX en de functies SIZE en INDEX.

Omdat het gaat om generieke pakketten, dus blauwdrukken voor bepaalde realisaties, moet bij een gegeven datatype een verschijningsvorm van het pakket worden gecreëerd ('instantiation'). Dit gebeurt met behulp van de generieke parameter ELEMENT_TYPE. Als we bijvoorbeeld I/O willen plegen met een paar numerieke typen dan kan dat als volgt:

```
with DIRECT_IO,SEQUENTIAL_IO;
procedure MAIN is
--
  package INTEGER_IO is new SEQUENTIAL_IO(ELEMENT_TYPE => INTEGER);
  use INTEGER_IO;
--
  package FLOAT_IO is new SEQUENTIAL_IO(ELEMENT_TYPE => FLOAT);
  use FLOAT_IO;
--
  type FIXED is delta 5.0 range -1_000.0 .. +5_000.0;
  package FIXED_IO is new DIRECT_IO(ELEMENT_TYPE => FIXED);
  use FIXED_IO;
--
begin
  -- body van MAIN
end MAIN;
```

SEQUENTIAL_IO en DIRECT_IO zijn voorgedefinieerde bibliotheekeenheden en moeten dus met behulp van een with-clausule worden geïmporteerd. Er werd weer van benoemde parameters gebruik gemaakt om de leesbaarheid van de verschijningsvorm te vergroten. Uit het voorbeeld blijkt verder, dat we niet alleen voor bestaande typen (INTEGER en FLOAT), maar ook voor door de gebruiker gedefinieerde typen (FIXED) een verschijningsvorm van een IO-pakket kunnen creëren. De use-clausule voegden we toe na elke creatie, opdat alle componenten van het pakket direct zichtbaar zijn en in het vervolg gebruikt kunnen worden.

Omdat alle datatypen volgens de regels van Ada gedeclareerd moeten worden, moet er ook een pakket worden gecreëerd voor elk type, dat we willen invoeren of weergeven via I/O-procedures. Alleen die routines die ook werkelijk nodig zijn behoeven daarbij te worden opgenomen en dit voorkomt onnodig extra geheugenbeslag bij ingebedde systeem toepassingen. Bij het ontwerp van Ada is uitgegaan van de gedachte dat programmatekst maar één keer geschreven wordt, maar dat deze honderd keren gelezen en begrepen moet worden. Vooral bij I/O-pakketten is in de programmatekst vaak wel erg veel omhaal van woorden nodig, maar de tekst laat aan duidelijkheid in ieder geval niets te wensen over. Als binnen een bepaalde toepassing een aantal I/O-pakketten regelmatig nodig blijkt, dan adviseren wij om er een pakket van te maken, waarin al de benodigde verschijningsvormen zijn opgenomen. Dit pakket kan dan in

zijn totaal met behulp van een with-clausule worden geïmporteerd waar het nodig is.

Bestandsstructuur

Alle invoer en weergave van gegevens in Ada, die niet op het allerlaagste en primitiefste niveau plaatsvinden, hebben te maken met bestanden of files. Een *file* heeft een uniek FILE_TYPE en is een theoretisch onbegrensde rij elementen. Elk element moet hetzelfde ELEMENT_TYPE hebben. Fysiek staat de file in verband met een of ander randapparaat, zoals een schijfeenheid, een terminal, een printer of een 'black box', zoals de voelers uit het vorige hoofdstuk. Logisch gezien wordt de I/O verwerkt via een file object. Alle bewerkingen op files zijn gedefinieerd voor objecten van het type FILE_TYPE. Uitgaande van het programma MAIN uit het laatste voorbeeld, kunnen we de volgende files declareren:

```
INTEGER_FILE : INTEGER_IO.FILE_TYPE;
FLOAT_FILE   : FLOAT_IO.FILE_TYPE;
FIXED_FILE   : FIXED_IO.FILE_TYPE;
```

De *modus* van een file wordt vastgesteld bij uitvoering van OPEN of CREATE. Voor bovenstaande pakketten kunnen de volgende drie modi worden gedefinieerd:

- IN_FILE
- OUT_FILE
- INOUT_FILE

De naam van de file modus (van het type FILE_MODUS) geeft de richting aan van de gegevensstroom ten opzichte van de gebruiker. Een file in modus IN_FILE, bijvoorbeeld, kan alleen worden gebruikt voor het invoeren van gegevens. Als een file eenmaal met een bepaalde modus is gedeclareerd (input, output of input/output), mag de modus tijdens de verwerking niet meer veranderd worden. Merk op dat de modus INOUT_FILE alleen bij DIRECT_IO files beschikbaar is.

Bij DIRECT_IO files wordt de file beschouwd als een verzameling elementen (alle van hetzelfde type) die een aantal opeenvolgende geheugenlokaties in beslag nemen. Waarden kunnen direct worden toegekend aan of opgevraagd van elk element van de file, op welke positie zich dit ook bevindt. Elke file heeft op ieder moment een eindig aantal elementen, voorgesteld door een geheeltallige waarde van het type COUNT. De file posities zijn genummerd van 1 tot aan de huidige omvang van de file: SIZE. De grootheid INDEX geeft de huidige lees/schrijfpositie aan en bij DIRECT_IO files kan deze positie worden ingesteld met behulp van SET_INDEX.

Bovenstaand model past goed bij bestanden op tape of disk, maar minder goed bij terminals, modems of 'black boxes' van allerlei soort. Hiervoor is het pakket SEQUENTIAL_IO geschikter. Bij SEQUENTIAL_IO files kan de positie niet worden geselecteerd, waarden worden verwerkt in de volgorde waarin zij verschijnen (vanuit het programma of vanuit de omgeving).

Om logische tegenspraak te voorkomen zijn de bestandsverwerkingsprocedures in Ada gedefinieerd, onafhankelijk van mogelijke fysieke beperkingen van de gegevensdragers. Sommige routines hebben daarom voor bepaalde fysieke files weinig zin: spatie-terug (backspace) op een modem is nu eenmaal niet mogelijk. Proberen we toch zo'n ongeschikte operatie toe te passen, dan zal de exceptie USE_ERROR ontstaan.

In het pakket IO_EXCEPTIONS zijn verder opgenomen:

DATA_ERROR	Ontstaat als een input operatie geen waarde van het correcte type oplevert (bijvoorbeeld als het systeem verwacht een integer waarde te lezen, maar in plaats daarvan een CHARACTER waarde ontvangt).
DEVICE_ERROR	Ontstaat als de (rand)apparatuur niet correct werkt.
END_ERROR	Ontstaat als we met READ voorbij het einde van de file willen lezen.
MODE_ERROR	Als we uit een OUT_FILE proberen te lezen of naar een IN_FILE proberen te schrijven.
NAME_ERROR	Als we een file proberen te creëren met CREATE of te OPENen met een verboden of niet unieke naam.
STATUS_ERROR	Als we een operatie proberen toe te passen op een file die niet open is, of als we een al open zijnde file proberen te OPENen.
USE_ERROR	Als we een operatie trachten uit te voeren, die op de gespecificeerde fysieke file niet is toegelaten.

Tenslotte is er nog de exceptie LAYOUT_ERROR voor TEXT_IO verwerking.

Bestandsverwerking

Voordat men met verwerking van gegevens via een file kan beginnen moet aan het fysieke bestand een naam worden toegekend met behulp van de subprogramma's CREATE en OPEN. Ook de modus van het bestand kan worden aangegeven. Met CREATE wordt een nieuw bestand gecreëerd en met OPEN komt een bestaand bestand voor verwerking beschikbaar. Dit gaat als volgt:

```
CREATE(FILE  => INTEGER_FILE,
        MODE => IN_FILE,
        NAME => "MIJN_DISK_FILE",
        FORM => "DISK2");
OPEN   (FILE => FLOAT_FILE,
        MODE => OUT_FILE,
        NAME => "MIJN_BLACK_BOX",
        FORM => "VOELERS");
```

De parameter NAME identificeert de naam van de file (welke namen zijn toegelaten is implementatie-afhankelijk). NAME kan verwijzen naar een bepaalde diskfile, of naar een of ander speciaal randapparaat. FORM is ook een implementatie-afhankelijke parameter en geeft bepaalde extra informatie over de file (bijvoorbeeld "NIET VERWIJDEREN").

Aan NAME en FORM kan niet alleen een letterlijke tekst worden toegekend, maar ook stringobjecten:

```
TITEL : STRING(1 .. 80);
. . .
GET(TITEL);    -- vraag de gebruiker om een naam
CREATE(FILE => FLOAT_FILE, NAME => TITEL, FORM => "DISK3);
```

Proberen we CREATE of OPEN op een file die al open is, dan ontstaat STATUS_ERROR. Als men een file probeert te openen en men is daartoe niet bevoegd, dan ontstaat NAME_ERROR.

De CREATE en OPEN procedures veronderstellen 'bij verstek' de waarde "" (de lege string) voor NAME en FORM. Verder is voor SEQUENTIAL_FILEs de default modus OUT_FILE en voor DIRECT-_FILEs is deze INOUT_FILE.

Als we klaar zijn met bestandsverwerking moet de file gesloten worden:

```
CLOSE(FIXED_FILE);
DELETE(INTEGER_FILE);
```

De verbinding tussen het logische en het fysieke bestand is nu verbroken, maar in het eerste voorbeeld blijft het fysieke bestand wel bestaan. Is de file al gesloten dan ontstaat de exceptie STATUS-_ERROR. In het tweede voorbeeld verdwijnt ook de fysieke file. De exceptie NAME_ERROR ontstaat als verwijderen niet mogelijk of niet toegelaten is.

Er bestaat nog een andere operatie op zowel DIRECT_IO als SEQUENTIAL_IO bestanden, namelijk de procedure RESET. RESET heeft als effect dat men weer aan het begin van de file terecht komt en vandaar opnieuw kan lezen of schrijven. RESET komt op hetzelfde neer als eerst CLOSE en dan OPEN uitvoeren, maar de verbinding tussen het logische en het fysieke bestand wordt niet verbroken.

Er zijn vijf functies in Ada om de status van een file te onderzoeken. Als enige parameter hebben alle vijf functies de filenaam nodig. Bijvoorbeeld:

NAME	Geeft als waarde de string met de filenaam.
FORM	Geeft de bij de file behorende form-string.
IS_OPEN	Retourneert de waarde TRUE als de file open is.
END_OF_FILE	TRUE als er geen volgende waarde meer uit de file gelezen kan worden.
MODE	Retourneert de modus van de file.

Voor DIRECT_IO files kent Ada verder nog de functies SIZE (die het huidige aantal elementen in de file retourneert) en INDEX (retourneert de waarde van de huidige lees/schrijfpositie). Aan de index kan alleen bij DIRECT_IO files een waarde worden toegekend met behulp van SET_INDEX. Een voorbeeld:

```
SET_INDEX(FIXED_FILE,TO => 137);
```

De functie INDEX zal nu de waarde 137 retourneren. Als men de index instelt op een positie voorbij het einde van de file, dan ontstaat er pas een exceptie als men probeert vanaf deze ongedefinieerde positie te lezen via READ.

Zowel bij DIRECT_IO als bij SEQUENTIAL_IO files kan men lezen en schrijven met behulp van de subprogramma's READ en WRITE. READ kan worden toegepast op een file van het type IN_FILE of INOUT_FILE, WRITE kan worden toegepast op files van het type OUT_FILE of eveneens INOUT_FILE. We vermelden nogmaals, dat alleen DIRECT_IO files de modus INOUT_FILE kunnen hebben. Voor lezen en schrijven moet de filenaam vermeld worden en een waarde of variabele van het juiste type:

```
READ(FILE =>INTEGER_FILE, ITEM =>MIJN_INTEGER);
WRITE(FILE =>FLOAT_FILE,  ITEM =>3.14159);
```

Als deze subprogramma's worden aangeroepen, dan worden de waarden in binaire vorm gelezen of geschreven. Als we bijvoorbeeld met een of ander uniek randapparaat communiceren, dan kan de binaire input of output met behulp van de representatiespecificatie de correcte vorm krijgen. We moeten hier nog aan toevoegen dat het DIRECT_IO pakket ook READ en WRITE subprogramma's met een INDEX parameter bevat, zodat de lees/schrijfpositie direct kan worden ingesteld.

Bij wijze van demonstratie van Ada's I/O faciliteiten behandelen we een eenvoudig bestandsverwerkingsprobleem. We lezen een rij geheeltallige grootheden uit een diskfile, berekenen het cumulatieve totaal en schrijven het eindtotaal naar een daartoe nieuw te creëren file:

```
declare
  package INTEGER_IO is new DIRECT_IO(ELEMENT_TYPE => INTEGER);
  use INTEGER_IO;
  INPUT_DATA   : INTEGER_IO.FILE_TYPE;
  OUTPUT_DATA  : INTEGER_IO.FILE_TYPE;
  WAARDE       : INTEGER;
  SOM          : INTEGER := 0;
begin
  OPEN  (INPUT_DATA,IN_FILE,"MIJN_DISK_INPUT_FILE");
  CREATE(OUTPUT_DATA,OUT_FILE,"MIJN_DISK_OUTPUT_FILE");
  while not END_OF_FILE(INPUT_DATA)
    loop
      READ(INPUT_DATA,WAARDE);
      SOM := SOM + WAARDE;
    end loop;
  WRITE(OUTPUT_DATA,SOM);
  CLOSE(INPUT_DATA);
  CLOSE(OUTPUT_DATA);
end;
```

Omdat het aantal elementen in de outputfile niet bekend is, wordt de while-constructie gecombineerd met de END_OF_FILE functie, gebruikt om de file te doorlopen.

19.2 Tekst Input/Output



Met behulp van de pakketten SEQUENTIAL_IO en DIRECT_IO hebben we eigenlijk alle operaties tot onze beschikking voor input/output van ieder gewenst type. Weliswaar wordt binnen ingebedde systemen vaak gecommuniceerd met speciale randapparatuur, maar desalniettemin is input en output in voor mensen leesbare vorm toch ook nog vaak wenselijk. Van de generieke pakketten kunnen gemakkelijk verschijningsvormen worden gemaakt voor elementen van het type CHARACTER, maar met name voor numerieke I/O in voor mensen leesbare vorm zouden wij onze eigen routines moeten schrijven. Omdat behoefte hieraan vrij algemeen is, heeft Ada een afzonderlijk (niet generiek) pakket TEXT_IO. Dit pakket bevat dezelfde functies als de generieke pakketten, maar verder ook nog subprogramma's voor bepaling van de layout van tekst.

Filestructuur

Het pakket TEXT_IO biedt faciliteiten voor invoeren en weergave van gegevens als deze louter bestaan uit ASCII tekens. Ook alle TEXT_IO verwerking verloopt via files en als het TEXT_IO pakket

wordt gebruikt dan worden automatisch standaard input- en outputfiles geopend bij het begin van de verwerking van het programma. In een interactieve omgeving is de default inputfile meestal het toetsenbord en de default outputfile het beeldscherm.

Natuurlijk kunnen we ook zelf via CREATE en OPEN onze eigen TEXT_IO files creëren en openen:

```
with TEXT_IO;
procedure MAIN is
  MIJN_INPUT   : TEXT_IO.FILE_TYPE;
  MIJN_OUTPUT  : TEXT_IO.FILE_TYPE;
begin
  -- body van MAIN
end MAIN;
```

Omdat TEXT_IO een bibliotheekeenheid is, moet dit pakket met behulp van de with-clausule worden geïmporteerd in het programma. Net als bij SEQUENTIAL_IO mogen alleen files van het type IN_FILE of OUT_FILE worden gecreëerd of geopend; tekstfiles van het type INOUT_FILE zijn niet toegelaten.

Evenals bij de generieke pakketten zijn er voor TEXT_IO de volgende operaties:

- CLOSE
- CREATE
- DELETE
- OPEN
- RESET

Omdat bij TEXT_IO van default files wordt uitgegaan komen de meeste tekstverwerkingsroutines in twee vormen voor: de eerste zonder filespecificatie en verbonden met de default file, de tweede met een expliciet vermelde logische filenaam. Via de functies STANDARD_INPUT en STANDARD_OUTPUT kunnen de namen van de default input- en outputfile worden opgevraagd. Via SET_INPUT en SET_OUTPUT kan tijdens de verwerking voor andere default files worden gekozen en de functies CURRENT_INPUT en CURRENT_OUTPUT identificeren de huidige default files.

Ook de functies IS_OPEN, END_OF_FILE, MODE, NAME en FORM zijn beschikbaar voor TEXT_IO files. Deze functies werken op dezelfde manier als bij de generieke pakketten.

File layout

DIRECT_IO files lijken op arrays van elementen. Ditzelfde geldt voor TEXT_IO files, maar aan deze laatste wordt nog een dimensie toegevoegd. TEXT_IO richt zich meestal op een regelgewijs werkend apparaat zoals een regeldrukker of een beeldscherm en TEXT_IO bevat daarom een aantal faciliteiten voor het regelen van de opmaak van de tekst. Een tekstfile bestaat uit een aantal pagina's en deze

zijn genummerd vanaf 1. Elke pagina bestaat uit een aantal regels en kan een variabele of een vaste lengte hebben. Elke regel bestaat verder uit een variabel of een vast aantal kolommen. De pagina, de regel op de pagina en de kolom in de regel worden bijgehouden in een object van het type COUNT. De precieze vorm is implementatie-afhankelijk. Als de gebruikte fysieke file qua vorm in strijd is met deze abstracte voorstelling, dan ontstaat na een niet toegelaten operatie de exceptie USE_ERROR.

Regellengte en paginalengte van een OUT_FILE kunnen worden ingesteld met behulp van de procedures SET_LINE_LENGTH en SET_PAGE_LENGTH. Bijvoorbeeld:

```
SET_PAGE_LENGTH(MIJN_INPUT,TO => 66);
```

Precieze specificaties van SET_PAGE_LENGTH en de andere hier besproken subprogramma's worden gegeven in Appendix C.

Proberen we met PUT meer dan 66 regels in te voeren, dan zal er na de zesenzestigste regel een 'einde-pagina' teken in de tekst worden geplaatst. Deze tekens kunnen we ook zelf expliciet in de tekst plaatsen door de paginalengte op UNBOUNDED te zetten (waarde 0). Een einde-regel teken kunnen we nu zo plaatsen:

```
PUT(ASCII.CR & ASCII.LF);  -- plaats "wagen terug" en
                           -- "nieuwe regel"
```

Met behulp van LINE_LENGTH en PAGE_LENGTH kunnen de huidige waarden van de parameters worden opgevraagd.

Behalve met PUT en GET, die natuurlijk de waarden van kolom, regel en paginanummer wijzigen, kunnen deze waarden ook expliciet worden ingesteld. In TEXT_IO zijn er twee subprogramma's voor de kolompositie:

- COL geeft de huidige kolomwaarde
- SET_COL stelt de huidige kolom in

Proberen we een kolom in te stellen voorbij de huidige regellengte dat ontstaat de exceptie LAYOUT_ERROR.

Er zijn ook subprogramma's in verband met de regelpositie:

- LINE geeft het huidige regelnummer
- SET_LINE stelt het huidige regelnummer in

De volgende procedures bewerkstelligen een overgang naar een nieuwe regel:

- NEW_LINE alleen voor OUT_FILE
- SKIP_LINE alleen voor IN_FILE

De functie END_OF_LINE levert de waarde TRUE als bij het lezen het einde van een regel bereikt is.

De volgende operaties hebben betrekking op het paginanummer:

- PAGE — geeft huidige paginanummer
- NEW_PAGE — alleen voor OUT_FILE: ga naar volgende pagina
- SKIP_PAGE — alleen voor IN_FILE: ga naar volgende pagina
- END_OF_PAGE — geeft waarde TRUE als einde pagina bereikt is (alleen voor IN_FILE)

We kunnen nu een eenvoudig voorbeeld geven van het overslaan van 7 regels in de default outputfile en vervolgens gaan staan in de 26-ste kolom:

```
NEW_LINE(SPACING => 7);
SET_COL (TO      => 26);
```

Invoeren en weergeven van tekens en strings

Alle bestandsbewerkingen binnen TEXT_IO worden gerealiseerd met behulp van de overladen subprogramma's PUT en GET. Voor lettertekens en rijen van lettertekens geeft PUT als output de gespecificeerde waarde, vanaf de huidige pagina, regel en kolom. GET leest vanaf dezelfde positie en slaat daarbij 'einde-regel' of 'einde-pagina' tekens over. Na een PUT of een GET operatie verwijzen kolomnummer, regelnummer en paginanummer naar de eerstvolgende positie na het laatst gelezen of geschreven teken. Dit geldt alleen voor begrensde files; is UNBOUNDED gebruikt dan moeten de begrenzingen expliciet met behulp van PUT worden aangegeven.

De PUT en GET operaties worden vaak gecombineerd met NEW_LINE en SKIP_LINE. Veronderstel bijvoorbeeld eens, dat de regellengte op 10 is ingesteld en dat regelnummer en kolomnummer beide gelijk 1 zijn. Beschouw nu de volgende operaties:

```
PUT("Wie ");
PUT("wind ");
PUT("zaait ");
NEW_LINE
PUT("zal storm ");
PUT("oogsten.");
NEW_LINE;
```

Het resultaat zal de volgende output zijn op de printer of op het beeldscherm:

```
Wie wind        -- regel 1
zaait,          -- regel 2
zal storm       -- regel 3
oogsten.        -- regel 4
```

Hoewel we maar twee NEW_LINE operaties gebruiken, krijgen we toch vier regels, omdat steeds automatisch naar een nieuwe regel wordt gegaan als de ingestelde regellengte wordt overschreden.

I/O voor andere datatypen

Vooral bij interactieve toepassingen komt het ook veel voor, dat numerieke gegevens of enumeratiewaarden moeten worden ingevoerd of weergegeven. Gebruiken we de standaardroutines uit SEQUENTIAL_IO of DIRECT_IO, dan verschijnen de resultaten in binaire vorm; zij zullen naar een voor mensen leesbare vorm moeten worden getransformeerd. Om hieraan tegemoet te komen zijn in Ada binnen TEXT_IO faciliteiten opgenomen voor I/O van numerieke data en enumeratiedata.

Ook hier worden de procedures PUT en GET gebruikt voor input en output. Omdat het in dit geval gaat om door de gebruiker gedefinieerde numerieke of enueratietypen, moet een verschijningsvorm van een generieke eenheid, die al in TEXT_IO is opgenomen, worden gecreëerd. Voor numerieke typen betekent dit creatie van één of meer van de volgende drie pakketten:

```
use TEXT_IO;
--
type INDEX is range 0 .. 100;
package INDEX_IO is new INTEGER_IO(INDEX);
--
type MASS is digits 10;
package MASS is new FLOAT_IO(MASS);
--
type LENGTH is delta 0.125 range 0.0 .. 10.0;
package LENGTH_IO        FIXED_IO(LENGTH);
```

Voor elk van deze drie generieke pakketten heeft TEXT_IO een overladen vorm van het subprogramma GET ter beschikking. De input kan in een vrij formaat worden gegeven, maar TEXT_IO kent ook een formaat met een tevoren vastgestelde lengte: WIDTH (het maximum aantal regels dat kan worden ingelezen). Wij verwijzen naar Appendix C voor de andere mogelijke vormen van GET. In het algemeen slaat GET alle spaties en controletekens over, en leest vervolgens tekens in, zolang de ingelezen string consistent blijft met het gespecificeerde type, of totdat WIDTH tekens zijn ingelezen. DATA_ERROR ontstaat als de string niet klopt met de syntax van het opgegeven type, en dus ook als er niets ingelezen kan worden.

Ook binnen PUT kunnen WIDTH en (voor integers) BASE worden opgegeven. Verder kan in het geval van een drijvende of vaste punt notatie met behulp van FORE en AFT het aantal cijfers voor en achter de decimale punt worden gespecificeerd. Het aantal cijfers van de exponent in het geval van drijvende punt notatie, wordt in EXP aangegeven. Hier volgen wat voorbeelden:

```
PUT(26);                                -- 26
PUT(26,WIDTH => 5);                     -- bbb26 (b stelt een spatie voor)
PUT(26,WIDTH => 5,BASE => 8);           -- 8#32#
PUT(3.14159,WIDTH => 7,FORE => 1,
            AFT   => 3,EXP  => 1);      -- 3.142e0
PUT(2.78159,WIDTH => 5,AFT  => 3);      -- 2.782
```

Als het totaal in de PUT operatie gespecificeerde aantal posities dat van WIDTH overschrijdt, dan wordt de expliciet opgegeven WIDTH genegeerd.

Voor enumeratietypen zijn binnen TEXT_IO soortgelijke faciliteiten opgenomen. Voor het enumeratietype moet een verschijningsvorm van het generieke pakket ENUMERATION_IO worden gecreeërd. Dit pakket bevat ook I/O mogelijkheden voor BOOLEAN typen. GET werkt hier met een vrij input formaat, maar in PUT zijn weer een paar layout parameters opgenomen. GET maakt bij enumeratie I/O geen verschil tussen grote en kleine letters. Bekijk de volgende declaraties:

```
use TEXT_IO;
type STATUS is (NORMAAL,WAARSCHUWING,ALARM);
package STATUS_IO is new ENUMERATION_IO(ENUM => STATUS);
```

PUT kan nu de volgende vormen aannemen:

```
    PUT(NORMAAL                          -- NORMAAL
    PUT(WAARSCHUWING,WIDTH => 15);       -- bbbWAARSCHUWING
    PUT(ALARM,WIDTH => 7,
              SET => LOWER_CASE);        -- bbalarm
```

19.3 Primitieve Input/Output



In bepaalde gevallen kunnen de faciliteiten die SEQUENTIAL_IO, DIRECT_IO en TEXT_IO ons bieden van een te hoog abstractieniveau zijn, vooral als het gaat om de communicatie met één of ander gespecialiseerd randapparaat. Een mogelijke oplossing is dan om met behulp van Ada's pakketmechanisme onze eigen I/O-routines te bouwen. Daarbij kunnen we van representatiespecificaties gebruik maken om bepaalde van de hardware afhankelijke adressen aan te geven.

Gebruikelijk is in dergelijke situaties, om elk randapparaat te behandelen als een parallelle taak, die in een pakket wordt ingebed.

Voor apparaten, die op een elementair niveau moeten worden bestuurd, kan men gebruik maken van het pakket LOW_LEVEL_IO, dat de procedures SEND_CONTROL en RECEIVE_CONTROL bevat. Beide procedures worden in Ada gedefinieerd met een tweetal sterk implementatie-afhankelijke parameters. De eerste parameter geeft de naam aan van het randapparaat en de tweede specificeert het type van de te-zenden of te ontvangen gegevens.

We hebben nu gezien, dat Ada een aantal I/O mogelijkheden bevat, opgenomen in de taal zelf. In het volgende hoofdstuk gaan we nader in op de gevolgen van Ada's uitbreidbaarheid en het verband met het construeren van grote systemen.

Oefeningen

1, Declareer een type PERSONEEL_RECORD met de componenten NAAM (een string van 40 tekens), ZIEKENFONDSNUMMER (een integer tussen 0 en 999_99_9999), en ADRES (een string van 80 tekens). Maak een verschijningsvorm van een pakket voor random access I/O voor het record en declareer een file PERSONEELSGEGEVENS.

2. Schrijf een subprogramma dat alle records uit PERSONEELSGEGEVENS leest (tot aan het einde van de file) en dat de records in een array zet. Het subprogramma moet als parameters dit array retourneren en een geheel getal dat het aantal records in de file aangeeft.

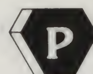
*3. Schrijf een subprogramma, dat de inhoud van het array uit opgave 2 naar een randapparaat schrijft, dat logisch gezien wordt voorgesteld door de file PRINTER. Druk NAAM af in de eerste 40 posities, sla vijf posities over, druk het ZIEKENFONDSNUMMER af in het opgegeven formaat, sla vijf posities over en druk vervolgens het ADRES af op de volgende regel. Sla twee regels over tussen elk record en druk 10 records per pagina af.

*4. Creëer een verschijningsvorm van een TEXT_IO pakket voor gegevens van het type INTEGER. Schrijf een subprogramma met als inputparameters ONDER, BOVEN en RESULTAAT. ONDER en BOVEN stellen de grenzen voor, waarbinnen de getallen moeten liggen en RESULTAAT is een geheel getal binnen deze grenzen. Probeer binnen het subprogramma via GET een RESULTAAT te lezen van de default inputfile. Gebruik exception handling om een DATA_ERROR op te vangen. PUT verder een toepasselijke boodschap als het RESULTAAT buiten de opgegeven grenzen valt en laat de gebruiker opnieuw input geven, zolang deze geen toegelaten waarde invoert.

20 PROGRAMMEREN IN HET GROOT

We hebben nu de meeste constructies in Ada uitgebreid bekeken en hun toepassing geïllustreerd met een aantal kleine voorbeelden. Ada werd echter niet ontworpen voor het oplossen van kleine problemen, maar juist voor problemen, waarvan de oplossing honderdduizenden - misschien zelfs miljoenen - programmaregels vergt. Het overgaan van het oplossen van kleine problemen naar het ontwikkelen van oplossingen voor grote problemen is niet alleen maar een kwestie van extrapoleren. Gelukkig zal in dit hoofdstuk blijken, dat Ada ons ook kan ondersteunen bij het ontwikkelen en onderhouden van grote en complexe systemen.

20.1 Het Geven Van Namen Aan Programma-Eenheden



Bij elke niet-triviale toepassing zal de oplossing bestaan uit een aantal programma-eenheden, zoals subprogramma's, pakketten en taken. Bij een groot systeem kan de oplossing worden onderverdeeld in honderden of zelfs duizenden eenheden. Meestal zal daarbij niet één, maar een team van programmeurs aan de ontwikkeling werken. Binnen de eenheden zullen objecten voorkomen, die onze abstractie van de werkelijkheid voorstellen en subprogramma's en taken zijn eveneens abstracties van operaties en acties. Voor al deze grootheden zijn betekenisvolle namen noodzakelijk.

In talen als FORTRAN en COBOL zouden we door de structuur van de taal gedwongen zijn de meest van die namen globaal te laten gelden voor het hele systeem. Dat betekent dat er bij in die talen geschreven projecten enorme lijsten (data dictionaries) nodig zijn, waarin de betekenis en het gebruik van alle namen in detail wordt beschreven. Telkens als een programmeur een bepaalde grootheid wil declareren of gebruiken, moet hij of zij eerst de data dictionary raadplegen, om na te gaan of de naam op de juiste manier gebruikt wordt, en of een naam niet dubbel gebruikt wordt. Omdat het daarbij grotendeels om globale, dus in het hele systeem bekende, namen zou gaan, moet in feite het hele systeem telkens opnieuw worden doorzocht of het effect van een of andere wijziging te bepalen. Het

domein van alle op een bepaalde plaats in het programma zichtbare namen, noemen we de *namenruimte*. Duidelijk is wel dat het beheren van de namenruimte in systemen, geschreven in FORTRAN of COBOL niet eenvoudig is.

Ook als we een groot systeem in Ada ontwikkelen hebben we honderden namen nodig, maar in Ada kunnen eenheden worden ingebed in blokken of pakketten. Op die manier is het mogelijk, alleen die grootheden op een bepaald punt in het programma zichtbaar te maken, die daar relevant zijn; alle onnodige details blijven verborgen. In de volgende paragraaf laten we zien, dat Ada een aantal vrij eenvoudige scope en zichtbaarheidsregels heeft, overeenkomstig aan de regels binnen ALGOL. Met behulp van deze regels is het mogelijk, de namenruimte te beheren.

Scope en zichtbaarheid

De *scope* van een grootheid is het gebied binnen het programma, waar de declaratie van die grootheid geldig is. Een grootheid is *zichtbaar* in het gebied, waar zijn naam gebruikt mag worden.

Een grootheid is altijd zichtbaar binnen het gebied van zijn scope en meestal is de grootheid dan ook binnen dit gehele gebied zichtbaar. In de volgende paragraaf zullen we grootheden tegenkomen, waarvan de namen overladen of verborgen zijn en die dus beperkt zichtbaar zijn, maar eerst bekijken we het eenvoudigste geval.

De scope van een naam begint op de plaats waar die naam gedeclareerd wordt (het eerst genoemd wordt) en strekt zich uit tot aan het einde van het subprogramma, pakket, taak of blok, waarin de declaratie is opgenomen. Worden namen in het specificatiegedeelte van een subprogramma, een pakket of een taak geïntroduceerd, dan loopt hun scope tot het einde van de bijbehorende body. Het omgekeerde is echter niet waar: de scope van een naam, die binnen een body wordt geïntroduceerd blijft beperkt tot die body en is in de specificatie van de eenheid niet zichtbaar.

Dit mag een beetje verwarrend lijken; laten we dus snel een paar voorbeelden bekijken, waarin de scope van de grootheden door grote haken wordt aangegeven:

```
procedure MAIN is
  --
  OBJECT_1 : INTEGER
  --
  procedure BINNENSTE_PROCEDURE is
    OBJECT_2 : INTEGER;
  begin
    -- body van BINNENSTE_PROCEDURE
  end BINNENSTE_PROCEDURE;
  --
begin
  BINNENSTE_BLOK :
  declare
    OBJECT_3 : INTEGER;
  begin
    -- body van BINNENSTE_BLOK
  end BINNENSTE_BLOK;
  -- rest van MAIN body
end MAIN;
```

De scope van OBJECT_1 strekt zich uit vanaf zijn declaratie tot aan het einde van MAIN. OBJECT_1 is binnen zijn gehele scope zichtbaar en we kunnen dus overal rechtstreeks naar OBJECT_1 verwijzen. De scope en de zichtbaarheid van OBJECT_2 zijn beperkt tot BINNENSTE_PROCEDURE en de scope en zichtbaarheid van OBJECT_3 zijn beperkt tot BINNENSTE_BLOK. Omdat OBJECT_1 zowel in BINNENSTE_PROCEDURE als in BINNENSTE_BLOK zichtbaar is, kan OBJECT_1 ook binnen de bodies van beide eenheden worden gebruikt.

OBJECT_1 is *globaal* ten opzichte van BINNENSTE_PROCEDURE en BINNENSTE_BLOK: OBJECT_2 en OBJECT_3 zijn *lokaal* ten opzichte van de eenheid, waarbinnen zij werden gedeclareerd. Men kan in Ada verwijzen naar globale objecten als deze zichtbaar zijn, maar het is niet mogelijk te verwijzen naar grootheden, die binnen een 'geneste' structuur, zoals een blok, worden gedeclareerd, van buiten die structuur. Men kan niet verwijzen naar een grootheid buiten de scope van die grootheid.

Met behulp van de puntnotatie kan wel naar de grootheden worden verwezen: BINNENSTE_PROCEDURE.OBJECT_2 en BINNENSTE_BLOK.OBJECT_3. In de volgende paragraaf zullen we zien, dat dit van pas komt in het geval van overladen of verborgen grootheden. Het is oòk aan te raden deze notatie te gebruiken als dit de betekenis kan verduidelijken.

In het laatste voorbeeld konden we direct met behulp van zijn naam naar een grootheid verwijzen; deze grootheid heet *direct zichtbaar*. De onderdelen van records, pakketten en taken zijn vaak niet direct zichtbaar. Dit wordt duidelijk in het volgende voorbeeld:

```
procedure MAIN is
  --
  type KNOOPPUNT is (SYSTEEM_1,SYSTEEM_2,SYSTEEM_3);
  --
  type PAKKET is
    record
      AFZENDER    : KNOOPPUNT;
      BESTEMMING  : KNOOPPUNT;
      BERICHT     : STRING(1 .. 100);
    end record;
  --
  MIJN_PAKKET : PAKKET;
  --
begin
  -- body van MAIN
end MAIN;
```

De grootheden KNOOPPUNT, PAKKET en MIJN_PAKKET hebben alle een scope en zijn alle zichtbaar vanaf hun introductie tot aan het einde van MAIN. De drie enumeratiewaarden SYSTEEM_1, SYSTEEM_2 en SYSTEEM_3 hebben dezelfde scope als KNOOPPUNT, maar de drie componenten van PAKKET: AFZENDER, BESTEMMING en BERICHT zijn niet direct zichtbaar. In de body van MAIN kan niet zomaar direct naar bijvoorbeeld AFZENDER verwezen worden. Hier moet van de puntnotatie gebruik worden gemaakt, dus binnen MAIN kunnen we het wel hebben over MIJN_PAKKET.AFZENDER, MIJN_PAKKET.BESTEMMING en MIJN_PAKKET.BERICHT.

Een soortgelijke regel geldt voor de componenten binnen pakketten en taken. Dit wordt in het voorbeeld boven aan bladzijde 334 geïllustreerd.

OBJECT_1 is in dit voorbeeld zichtbaar door geheel MAIN en kan zelfs binnen MIJN_PACKAGE en MIJN_TASK direct genoemd worden. De scope van OBJECT_2 beslaat ook de body van MIJN-_PACKAGE, maar OBJECT_3 is alleen binnen deze body zichtbaar. Omdat OBJECT_2 en BINNENSTE_PROCEDURE in de specificatie van MIJN_PACKAGE worden ingevoerd, kunnen ze binnen de scope van MIJN_PACKAGE worden gebruikt. Van buiten MIJN_PACKAGE kunnen we echter niet direct OBJECT_2 of BINNENSTE_PROCEDURE bereiken; dit moet weer met de puntnotatie. In MAIN kunnen we het wel hebben over MIJN_PACKAGE.OBJECT_2 en over MIJN_PACKAGE. BINNENSTE_PROCEDURE en evenzo mogen we spreken van MIJN-_TASK.ROUTE_1 en MIJN_TASK.ROUTE_2.

```
procedure MAIN is
  --
  OBJECT_1 : INTEGER;
  --
  package MIJN_PACKAGE is
    OBJECT_2 : INTEGER;
    procedure BINNENSTE_PROCEDURE;
  end MIJN_PACKAGE;
  --
  package body MIJN_PACKAGE is
    OBJECT_3 : INTEGER;
    -- rest van de body van MIJN_PACKAGE
  end MIJN_PACKAGE;
  --
  task MIJN_TASK is
    entry ROUTE_1;
    entry ROUTE_2;
  end MIJN_TASK;
  task body MIJN_TASK is
    -- body van MIJN_TASK
  end MIJN_TASK;
  --
begin
  -- body van MAIN
end MAIN;
```

Als er pakketten binnen pakketten voorkomen, dan kunnen verwijzingen van buiten met behulp van de puntnotatie nogal lang worden. Een manier om dat te omzeilen is gebruik te maken van de use-clausule. Nadat een pakketspecificatie is gegeven, of nadat een contextspecificatie met een with-clausule is vermeld, kunnen we zeggen:

```
use MIJN_PACKAGE;
```

Nu hoeft de puntnotatie niet meer te worden gebruikt, tenzij er door de pakketnaam als voorvoegsel weg te laten logische tegenspraken wegens meervoudig voorkomende namen zouden kunnen ontstaan.

We willen wel waarschuwen voor het gebruik van de use-clausule. Vooral als het aantal grootheden in de naamruimte toeneemt is het gevaar voor inwendige logische tegenspraak niet denkbeeldig. Voor taken bestaat er niet zoiets als de use-clausule: bij taken zullen we dus altijd van de puntnotatie gebruik moeten maken om taakcomponenten te kunnen benaderen.

Overladen, verbergen en herbenoemen

Bij het ontwikkelen van een systeem willen we ons geen zorgen hoeven te maken over eventuele logische conflicten tussen namen. Liefst geven we elke grootheid een naam, die de betekenis of het gebruik van die grootheid omschrijft. Vooral in grote systemen kan dat wel eens betekenen, dat twee verschillende grootheden logisch gezien dezelfde naam zouden moeten hebben. Ook hiervoor biedt Ada een oplossing, zoals we zullen zien.

Als dezelfde naam of hetzelfde operatorsymbool voor twee verschillende grootheden wordt gebruikt, dan heet die naam of dat symbool *overladen*. Bijvoorbeeld:

```
declare
  type KLEUR is (ROOD,GROEN,BLAUW);
  type LICHT is (ROOD,GEEL,GROEN);
  BEELDPUNT : KLEUR;
  STOPLICHT : LICHT;
begin
  -- body van dit blok
end;
```

ROOD en GROEN zijn beide overladen, want zij worden binnen twee verschillende omgevingen gebruikt. We mogen in Ada dergelijke overladen namen vrijelijk gebruiken, zolang maar duidelijk is welk ROOD of GROEN we bedoelen. We kunnen dus schrijven:

```
BEELDPUNT := GROEN;
STOPLICHT := GROEN;
```

In beide gevallen kan de compiler uit de context opmaken over welk GROEN het gaat, door het linkerlid van de waardetoekenningsopdracht te analyseren. Als mogelijke tegenstrijdigheid niet door de context wordt opgeheven, dan kan altijd van een kwalificatie expressie gebruik worden gemaakt:

```
BEELDPUNT := KLEUR'(ROOD);
```

Ook andere grootheden dan enumeratiewaarden kunnen overladen worden: aggregaten, numerieke waarden, subprogramma's en taakentries. Zo kunnen wij bijvoorbeeld een subprogramma PUT als volgt overladen:

```
procedure PUT(ELEMENT : INTEGER);
procedure PUT(ELEMENT : FLOAT);
procedure PUT(ELEMENT : KLEUR);
procedure PUT(ELEMENT : LICHT);
```

De keuze om een subprogramma op deze wijze te overladen ligt voor de hand: zolang het om logisch gezien dezelfde operatie of

hetzelfde object gaat, is er geen reden dit een andere naam te geven. Natuurlijk moet het overladen wel met overleg gebeuren: de begrijpelijkheid kan er door in het gedrang komen.

We kunnen elk van bovenstaande subprogramma's aanroepen: op grond van de taalregels kan steeds worden bepaald om welke PUT het gaat:

```
PUT(367);                 -- PUT een INTEGER
PUT(4.6);                 -- PUT een FLOATing point getal
PUT(BLAUW);               -- PUT een kleur
PUT(LICHT'(GROEN));       -- PUT waarde van LICHT
```

In het laatste geval is het noodzakelijk een gekwalificeerde expressie te gebruiken: PUT(GROEN) zou dubbelzinnig zijn. Ook de puntnotatie kan gebruikt worden om dergelijke dubbelzinnigheden te elimineren bij het gebruik van overladen grootheden.

Een naam heet dus overladen als deze wordt gebruikt voor meer dan één grootheid. Sommige grootheden, met name objecten, kunnen echter niet worden overladen. Zodra een naam in een declaratie binnen een geneste structuur wordt gebruikt, is elke andere grootheid buiten deze structuur met dezelfde naam verborgen. Bekijk bijvoorbeeld eens het volgende stukje programma:

```
procedure MAIN is
  --
  MIJN_OBJECT : BOOLEAN;              ─────────┐
  --                                           │
begin                                          │
  BINNENSTE_BLOK;                              │
  declare                                      │
    MIJN_OBJECT : BOOLEAN;            ──┐      │
  begin                                 │      │
    -- body van BINNENSTE_BLOK          │      │
  end BINNENSTE_BLOK;                 ──┘      │
  -- rest van body van MAIN                    │
end MAIN;                             ─────────┘
```

Hier heeft MAIN.MIJN_OBJECT een scope vanaf het punt van declaratie tot aan het einde van MAIN. De scope van BINNENSTE-_BLOK.MIJN_OBJECT is beperkt tot dit blok zelf. Hebben we het in het BINNENSTE_BLOK over MIJN_OBJECT, dan wordt door Ada automatisch aangenomen dat we het lokale object bedoelen. De globale grootheid kan vanuit dit blok alleen bereikt worden via MAIN.MIJN_OBJECT.

In het algemeen zal het verborgen zijn van globale grootheden, als er een lokale grootheid met dezelfde naam wordt gebruikt, geen problemen opleveren. Directe verwijzing naar globale grootheden zal weinig voorkomen en als dit wel nodig is, dan is het verstandiger zo'n grootheid bijvoorbeeld als een subprogrammaparameter te importeren. Als we onze oplossingen een gelaagde structuur geven,

met steeds diepere niveaus van afnemende abstractie, dan hoeft het ons meestal niet te interesseren welke namen er op de hogere niveaus werden gebezigd.

Wel zagen we al dat bij gebruik van de puntnotatie, om namen binnen geneste structuren te bereiken, de uiteindelijke omschrijvingen nogal lang kunnen worden. Een manier om dit te voorkomen is, behalve de al genoemde use-clausule, ook het herbenoemen van pakketten. Laten we eens uitgaan van de volgende geneste pakketdeclaratie:

```
package MATH_FUNCTIES is
  --
  package TRIG_FUNCTIES is
    function COS(HOEK : RADIALEN) return FLOAT;
  end TRIG_FUNCTIES;
  package MATRIX is
    --
  end MATRIX;
end MATH_FUNCTIES;
```

Als we niet de use-clausule willen gebruiken, omdat deze de namenruimte vergroot en daardoor de kans op dubbelzinnigheid doet toenemen, zouden we naar de COS functies als volgt moeten verwijzen:

```
MATH_FUNCTIES.TRIG_FUNCTIES.COS(EEN_HOEK);
```

We kunnen het binnenste pakket echter als volgt herbenoemen:

```
package TRIG renames MATH_FUNCTIES.TRIG_FUNCTIES;
```

Nu kunnen we het hebben over:

```
TRIG.COS(EEN_HOEK);
```

en dat is aanmerkelijk korter.

Ook objecten, excepties, taakentries en subprogramma's kunnen herbenoemd worden. Conflicten tussen namen kunnen op die manier opgelost worden en het is ook mogelijk één grootheid onder twee verschillende namen bekend te laten staan. Als we de volgende declaraties hebben:

```
MIJN_ARRAY : array (1 .. 10) of INTEGER;
ALARM      : exception;
procedure QUICK_SORT(ELEMENTEN : in out LIJST) is ...
```

Dan kunnen deze grootheden bijvoorbeeld zo worden herbenoemd:

```
UW_INTEGER : INTEGER renames MIJN_ARRAY(3);
WATER_NIVEAU : exception renames ALARM;
procedure SORT(X : in out LIJST) renames QUICK_SORT;
```

Bij het array-element wordt het herbenoemen gebruikt om de array-index maar één keer te hoeven evalueren. Bij het herbenoemen van het subprogramma QUICK_SORT is te zien, dat ook de benaming van de formele parameters gewijzigd mag worden, en ook de default waarden (voorzover aanwezig) mogen worden aangepast.

Als we een andere naam aan een type willen geven dan kan dat behalve via herbenoemen ook door het invoeren van een subtype:

```
package ABSTRACT_TYPE is
  --
  type MIJN_TYPE is ...
  --
end ABSTRACT_TYPE;
package body ABSTRACT_TYPE is ...
--
subtype LOKAAL_TYPE is ABSTRACT_TYPE.MIJN_TYPE;
```

Een waarschuwing tegen het gebruik van herbenoemen via 'renames' is op zijn plaats: er kan nu op meer dan één manier naar dezelfde variabele worden verwezen en dit staat bekend als *aliasing*. De ene programmeur zou de ene naam voor een bepaald object kunnen gebruiken en een andere programmeur een andere naam voor hetzelfde object; wat de gevolgen zijn bij het toekennen van waarden laten wij aan de fantasie van de lezer over.

Bij de bouw van grote systemen zal het aantal namen in de namenruimte steeds groter worden. Door middel van herbenoemen kan men kortere namen gebruiken, maar het aantal namen neemt er niet door af, in tegendeel. Een methode, die ons wel kan helpen bij het beperkt houden van het aantal namen op ieder niveau, is de afzonderlijke compilatie van programma-eenheden. Daarover in de volgende paragraaf meer.

20.2 Afzonderlijke Compilatie



Bij elk softwaresysteem van enige omvang is het niet alleen verstandig, maar zelfs noodzakelijk om de programma-eenheden zo min mogelijk onderling afhankelijk te laten zijn. Bij de ontwikkeling van een groot systeem zijn vaak verschillende teams betrokken, die elk voor een deel van de totale oplossing verantwoordelijk zijn. Elk onderdeel kan daarom een een verschillend stadium van ontwikkeling zijn en het testen van deze onderdelen is alleen maar mogelijk als zij inderdaad als afzonderlijke eenheden aan een onderzoek kunnen worden onderworpen. Het zou trouwens ook nogal onhandig zijn als het gehele systeem telkens opnieuw gecompileerd zou moeten worden als ergens in een onderdeel een wijziging wordt aangebracht.

Kortom, we willen de systeemmodulen afzonderlijk kunnen compileren. Dit kan ook wel in andere programmeertalen, maar Ada biedt nog wat extra gereedschap voor het beheersen van het ontwikkelingsproces van afzonderlijk gecompileerde modulen.

Bibliotheekeenheden

We hoeven in Ada niet in één keer een heel programma te compileren, hoewel dat natuurlijk wel kan als we het per se willen. In Ada kan de programmatekst aan meer dan één compilatie worden onderworpen. Een compilatie bestaat uit één of meer compilatie-eenheden, te weten:

- generieke declaratie
- generieke verschijningsvorm creatie (instantiatie)
- subprogrammadeclaratie
- subprogrammabody
- pakketdeclaratie
- pakketbody
- subunit

Een taak is geen compilatie-eenheid, daarom wordt een taak vaak ingebed in een pakket om er toch een compilatie-eenheid van te kunnen maken.

De compilatie-eenheden van een programma behoren tot de zogenaamde *programmabibliotheek*. Elke compilatie-eenheid is ook een *bibliotheekeenheid*. Alle programma-eenheden dienen unieke namen te bezitten; ook het hoofdprogramma is zo'n eenheid, maar de identificatie daarvan wordt aan de programmeeromgeving overgelaten (zie hoofdstuk 22). Als een stuk programmatekst voor compilatie wordt aangeboden, dan wordt dit door Ada zelf behandeld, alsof dit binnen de context van een pakket STANDARD (zie Appendix D) is gedeclareerd. Alle grootheden uit STANDARD kunnen dus, tenzij hun namen verborgen zijn, direct worden gebruikt.

Eenheden kunnen met behulp van de **with**-clausule toegang verkrijgen tot eerder gecompileerde bibliotheekeenheden. Terugkomend op ons allereerste ontwerpprobleem uit hoofdstuk 7 kunnen de eenheden BLAADJES_PAKKET, VERZAMELING_PAKKET en BOOM_PAKKET afzonderlijk voor compilatie worden aangeboden. Een volgende eenheid kan naar deze eenheden in een contextspecificatie verwijzen:

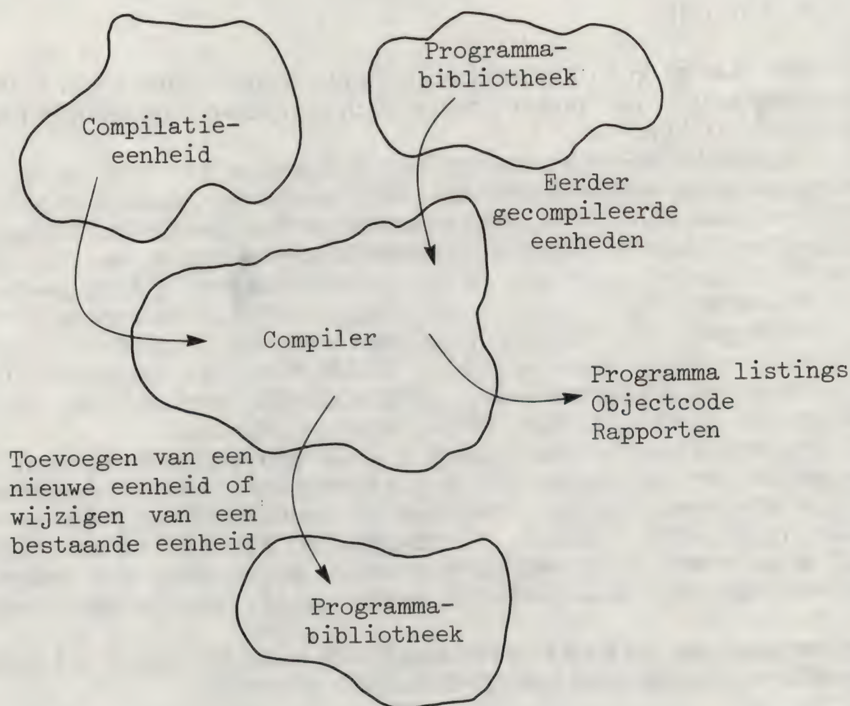
```
with BLAADJES_PAKKET,VERZAMELING_PAKKET,BOOM_PAKKET;
procedure TEL_BLAADJES_IN_BINAIRE_BOOM is ...
```

In het vervolg kan nu de puntnotatie gebruikt worden om de componenten van deze pakketten te bereiken, of men kan gebruik maken van de **use**-clausule. Zo kunnen we dus precies die grootheden

toegankelijk maken, die we nodig hebben, niet meer en niet minder. Bij alle bibliotheekeenheden, ook bij generieke eenheden, kan de with-clausule worden toegepast. Wel wordt door dit soort context-bepalingen de volgorde van compilatie van de eenheden grotendeels vastgelegd.

Het idee van afzonderlijke compilatie werd ook in andere programmeertalen toegepast, maar de regels van Ada zorgen ervoor dat in de ene module gedeclareerde datatypen in een andere module als zodanig erkend blijven. De voordelen van strenge datatypering blijven zo behouden en hoewel dit de Ada compiler er zeker niet eenvoudiger op maakt, is het vooral bij het ontwikkelen van grote systemen een zeer belangrijke eigenschap.

Zoals blijkt uit figuur 20-1 gaan Ada compilers uit van het bestaan van een programmabibliotheek [1]. Te compileren eenheden kunnen in een contextspecificatie het gebruik van bibliotheekeenheden specificeren. De gecompileerde eenheid zelf wordt, als de compilatie succesvol was, op zijn beurt een onderdeel van de bibliotheek en kan dus weer door andere eenheden worden gebruikt. In hoofdstuk 22 zullen we nog enige hulpmiddelen bespreken voor het beheer van deze programma-eenheden bibliotheek.



Figuur 20-1 Model voor compilatie in Ada.

Subeenheden

Bij het top-down ontwikkelen van systemen komt het vaak voor, dat de elementaire bouwstenen, waaruit zo'n systeem uiteindelijk wordt samengesteld, nog niet allemaal gereed zijn. In dat geval kunnen voorlopige stoplappen of 'stubs' worden gebruikt, die afzonderlijk kunnen worden gecompileerd. In Ada heten deze stubs *subeenheden*.

Onderstaand hoofdprogramma hebben we naar functie onderverdeeld:

```
procedure MAIN is
  task BLACK_BOX is
    entry ONTVANG(BERICHT : out STRING);
    entry ZEND    (BERICHT : in  STRING);
  end BLACK_BOX;
  task body BLACK_BOX is separate;
  --
  package TRANSFORMEER is
    procedure ONTCIJFER(BERICHT : in out STRING);
    procedure CODEER   (BERICHT : in out STRING);
  end TRANSFORMEER;
  package body TRANSFORMEER is separate;
  --
  procedure RAPPORTEER(BERICHT : in STRING is separate);
  --
begin
  -- body van MAIN
end MAIN;
```

De taak-, de pakket- en de subprogrammabody kunnen nu op de volgende manier afzonderlijk worden gecompileerd:

```
separate (MAIN)
task body BLACK_BOX is ...

separate (MAIN)
package body TRANSFORMEER is ...

separate (MAIN)
procedure RAPPORTEER(BERICHT : in STRING) is ...
```

In het hoofdprogramma worden dus 'stubs' geplaatst van de vorm 'task body BLACK_BOX is separate' en deze kunnen later worden uitgewerkt op de bovenstaande manier. (Misschien ten overvloede: op de plaats van de puntjes moet de werkelijke body worden ingevuld!) De 'separate (MAIN)'-clausule is noodzakelijk om de 'ouder' van de afzonderlijke bodies te kunnen identificeren. Net zoals eerder bij de with-clausule, wordt hier een compilatie- en definitievolgorde binnen de programma-eenheden geïntroduceerd. Tenslotte: ook binnen de subeenheden kunnen weer bibliotheekeenheden met behulp van with-clausules worden gebruikt.

Volgorde van compilatie en hercompilatie

Als een systeem in meerdere compilatiegangen wordt opgebouwd, dan volgt uit onderlinge afhankelijkheden tussen eenheden ook een bepaalde compilatievolgorde. Een eenheid moet zijn gecompileerd om voor een andere eenheid zichtbaar te kunnen zijn. Zo moet de specificatie van een subprogramma, een pakket of een taak zijn gecompileerd vóórdat de bijbehorende body kan worden gecompileerd. Een 'ouder' eenheid moet worden gecompileerd vóór zijn subeenheden. Worden via een **with**-clausule bibliotheekeenheden genoemd, dan moeten deze eveneens zijn gecompileerd. Onafhankelijke pakketbodies kunnen in willekeurige volgorde worden gecompileerd, is er wel afhankelijkheid, dan zal een goede compiler ons waarschuwen als we de compilatie in een niet toegelaten volgorde proberen uit te voeren.

Tijdens het ontwikkelen van een systeem zal het ongetwijfeld voorkomen, dat een eenheid opnieuw moet worden gecompileerd, bijvoorbeeld na correctie van een fout, of na wijziging van een algoritme. Door Ada's compilatiefaciliteiten kan dit zonder enig probleem: alleen de gewijzigde eenheid hoeft opnieuw te worden gecompileerd. Zolang we het specificatiegedeelte van een eenheid niet veranderen, mogen we het implementatiegedeelte (de body) naar hartelust veranderen en opnieuw compileren. De logische structuur van het ontwerp blijft zo behouden; effecten van wijzigingen blijven plaatselijk.

Alleen als we een 'ouder' van een subeenheid wijzigen, moeten we ook die subeenheid opnieuw compileren. Ook als de specificatie van een bibliotheekeenheid wordt gewijzigd heeft dit verder strekkenden gevolgen: niet alleen moet de body van de eenheid opnieuw worden gecompileerd, ook alle eenheden, die van deze bibliotheekeenheid met een **with**-clausule gebruik maken, moeten opnieuw worden gecompileerd. In hoofdstuk 22 zullen we zien, dat de Ada programmeeromgeving ons behulpzaam kan zijn bij het aangeven van de opnieuw te compileren eenheden na een wijziging.

20.3 De Architectuur Van Grote Systemen



Al in hoofdstuk 4 wezen we erop dat een rommelige en ongedisciplineerde ontwerptechniek de kans op het succesvol ontwikkelen van een groot softwaresysteem erg klein maakt. Grote systemen worden vrijwel nooit in één keer opgebouwd; zij komen voort uit kleinere systemen. Eigenlijk moet de ontwikkeling van een softwaresysteem gelijke tred houden met de ontwikkeling van onze kijk op het probleemgebied in de realiteit. Door van Ada's uitgebreide verzameling van mogelijkheden voor het creëren van afzonderlijk compileerbare programma-eenheden gebruik te maken, kan dit ook, en de ontwikkeling kan zowel top-down als bottom-up plaatsvinden.

Top-down ontwikkeling

Bij top-down ontwerp begint men op het hoogste niveau van abstractie en daalt men stapsgewijs af tot lagere en meer primitieve niveaus. Yourdon's functionele ontwerpmethode is zo'n top-down methode (zie nog eens hoofdstuk 4).

Ada's subeenheden kunnen bij top-down ontwikkeling worden toegepast. Het hoogste niveau van een programma kan er bijvoorbeeld zo uitzien:

```
procedure MAIN is
  procedure INPUT  (D : out    DATA) is separate;
  procedure PROCES (D : in out DATA) is separate;
  procedure OUTPUT (D : in     DATA) is separate;
begin
  -- body van MAIN
end MAIN;
```

De procedures worden als stubs aangebracht en kunnen naderhand nader worden ingevuld en afzonderlijk worden gecompileerd:

```
separate (MAIN)
procedure INPUT ...

separate (MAIN)
procedure PROCES ...

separate (MAIN)
procedure OUTPUT ...
```

Bij verdere opdeling van deze eenheden kunnen nieuwe subeenheden worden gedeclareerd.

Bij de meeste systeemontwikkelingen wordt een top-down ontwerpmethode toegepast en deze methode kan met behulp van Ada ook heel goed worden uitgevoerd. Alleen als een aantal programma-eenheden van dezelfde module gebruik moet maken, dan is deze methode minder geschikt, omdat immers elke subeenheid zijn 'ouder' of voortbrengende eenheid expliciet moet noemen en dit wordt lastig als er sprake is van meer dan één ouder.

Bottom-up ontwikkeling

Bij een bottom-up ontwikkeling begint men met het maken van elementaire bouwstenen, die later een een groot aantal modules kunnen worden gebruikt. Zo'n bouwsteen kan bijvoorbeeld het volgende pakket zijn, dat een abstract datatype COMPLEX.GETAL exporteert:

```
package COMPLEX is
  type GETAL is ...
  -- abstracte operaties op GETAL objecten
end COMPLEX;
```

Naar deze module kunnen we verwijzen in een contextspecificatie en vervolgens kunnen we van de mogelijkheden, die het pakket biedt, gebruik maken:

```
with COMPLEX;
procedure MAIN is
  MIJN_GETAL : COMPLEX.GETAL;
begin
  -- body van MAIN
end MAIN;
```

Bij de ontwikkeling van grote systemen wordt meestal zowel de top-down als de bottom-up methode gebruikt. Dat deden wij ook in de door ons voorgestelde objectgerichte ontwerpmethode: we identificeren objecten en operaties van bovenaf, maar we gebruiken elementaire faciliteiten om ze uit te werken. In het volgende hoofdstuk behandelen we een groot ontwerpprobleem, waarbij van beide methodes gebruik wordt gemaakt en waarin veel geavanceerde mogelijkheden van Ada opnieuw aan bod komen.

Oefeningen

1. Laat nog eens de vier ontwerpproblemen die we behandelden de revue passeren en geef aan welke ontwerpmethode of combinatie van ontwerpmethodes we gebruikten.

2. Wat zou het gevolg zijn, als afzonderlijke compilatie in Ada zou gebeuren zonder stringente typecontrole tussen programma-eenheden onderling?

3. Een eenheid in Ada is alleen binnen zijn scope zichtbaar. Welke taalmogelijkheid maakt verwerking buiten de scope mogelijk?

4. Geef overeenkomsten en verschillen aan tussen *overladen* en *verbergen*.

21 VIJFDE ONTWERPPROBLEEM: 'KOP OP' DISPLAY

Eén van de redenen voor de softwarecrisis is, dat de op te lossen problemen dermate omvangrijk en ingewikkeld zijn geworden, dat het overzien van de oplossing voor het totale probleem onze intellectuele capaciteiten te boven gaat. (In hoofdstuk 4 wezen we daar al op.) Of één enkele programmeur een programma van een miljoen regels tot in alle details kan overzien, valt zeer te betwijfelen. De zogenaamde Hrair limiet wordt hier duidelijk overschreden. Het onderhouden en aanpassen van een dergelijk programma zonder de structuur te niet te doen is opnieuw een zeer moeilijke taak.

In de laatste paar hoofdstukken hebben we de bijzondere eigenschappen van Ada onderzocht en hebben we de in Ada mogelijke constructies uitgebreid bestudeerd. We hebben laten zien dat Ada is gebaseerd op moderne ideeën omtrent programmeren, zoals het gebruik van abstracte datastructuren en van 'information hiding'. Juist omdat Ada dergelijke methodes ondersteunt, is het meer dan alweer een nieuwe programmeertaal; Ada biedt een verzameling gereedschappen, die behulpzaam kunnen zijn bij het beheersen van de complexiteit van oplossingen.

Misschien is ook bij het gebruik van Ada een buitengewoon groot programma niet tot in alle details te begrijpen, maar in ieder geval kan de oplossing in Ada op een heldere en modulaire manier worden geformuleerd. Men kan nu de onderdelen afzonderlijk bestuderen en op grond van begrip van de onderdelen uiteindelijk ook het totaal begrijpen. Natuurlijk geldt ditzelfde ook voor kleinere problemen; waar het om gaat is, dat Ada helpt bij het helder onder woorden brengen van de oplossingsmethode, terwijl het uiteindelijke programma deze oplossing op een efficiënte wijze kan realiseren.

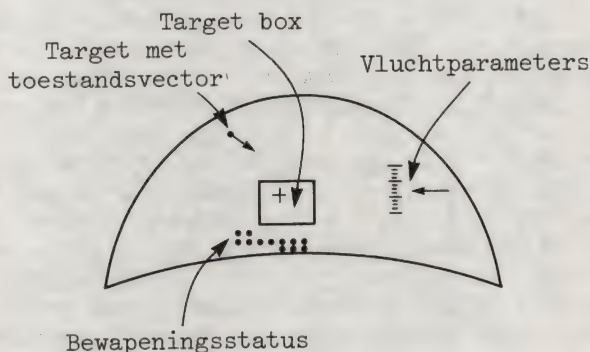
We behandelden tot nu toe vier ontwerpproblemen met een toenemende graad van moeilijkheid. Omdat Ada vooral is ontworpen voor ingebedde computertoepassingen zullen we nu nagaan hoe Ada kan worden gebruikt bij het oplossen van een groot probleem, waarbij de communicatie met andere systemen de randvoorwaarden bepaalt. Omdat de totale oplossing uit een duizendtal programmaregels zal bestaan, zullen we niet de hele oplossing presenteren. We zullen ons beperken tot het ontwerp van de globale structuur en we zullen alleen het hoogste abstractieniveau nader uitwerken.

21.1 Definieer Het Probleem

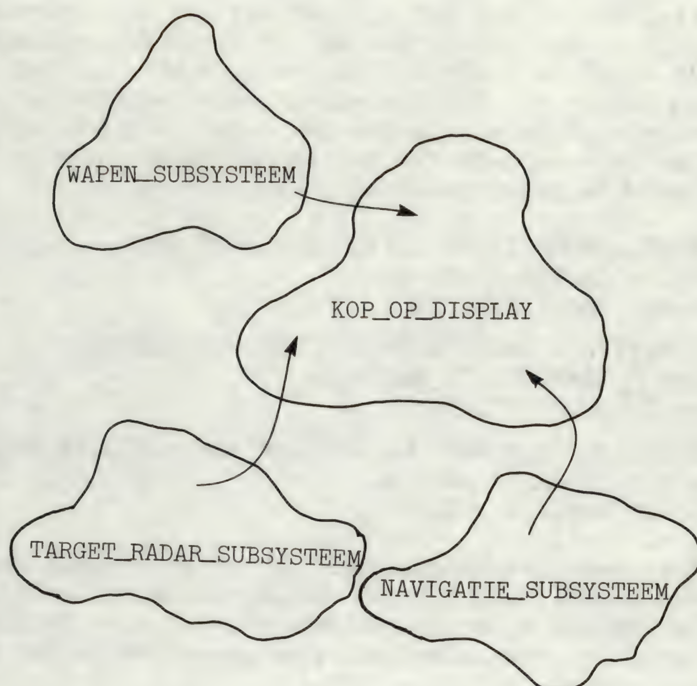


Bij het benaderen van een doel door een modern militair vliegtuig is het van het grootste belang dat de mens-machine communicatie zo eenvoudig mogelijk verloopt. Er is eenvoudig geen tijd om in de cockpit rond te kijken ter bestudering van de instrumenten: de piloot moet voortdurend het doel in het oog houden. Dit probleem kan worden opgelost door een 'kop op' beeldscherm te ontwikkelen (Engels: 'heads-up display' of HUD). Op dit beeldscherm moet de piloot alle belangrijke gegevens omtrent zijn doel tegelijk met de belangrijkste vluchtparameters in één oogopslag kunnen zien. Bij de meeste HUDs worden deze gegevens geprojecteerd op het windscherm voor hem, zodat de piloot tegelijkertijd ook naar buiten kan blijven kijken. In figuur 21-1 is het beeld dat de piloot ziet schematisch weergegeven.

Het doel van het display is het verschaffen van voldoende informatie bij een zo laag mogelijk complexiteitsniveau. De piloot moet vliegen volgens een scenario, waarbij hij ervoor zorgt dat het doel binnen de target box (doelvierkant) terechtkomt. Als hij op dat moment vuurt, wordt het doel zeker geraakt. Als de piloot gebruik maakt van een type boordwapen, dat binnen een zeker bereik automatisch gericht kan worden, dan wijst de cursor binnen de box aan hoe het wapen gericht staat. Verder zal de target box, afhankelijk van het bereik van het gekozen wapen, in omvang toenemen naarmate het vliegtuig dichter bij het doel komt. De display-eenheid geeft ook een aantal belangrijke vluchtparameters weer, zoals hoogte, aanvalshoek en een overzicht van de bewapeningsstatus. Over het doel zelf wordt een pijltje geprojecteerd. Dit is niet alleen bedoeld om het juiste doel aan te geven, maar ook om de waarschijnlijke richting van het doel te voorspellen.



Figuur 21-1 Zoals de piloot een 'heads-up display' ziet.



Figuur 21-2 het 'Kop-op' display en zijn subsystemen.

Een zo complex systeem als een vliegtuig zal een groot aantal ingebedde deelsystemen bevatten. Als we onze HUD ontwerpen, zullen we ongetwijfeld rekening moeten houden met een aantal randvoorwaarden, die ons zullen worden voorgeschreven en waarop wij geen invloed hebben: we kunnen nu eenmaal niet het hele vliegtuig opnieuw ontwerpen om het bij onze wensen aan te passen. Zelfs voor we beginnen met het ontwerp van ons systeem, hebben andere ontwerpteams hoogstwaarschijnlijk al andere deelsystemen gedefinieerd, zoals de automatische piloot of het radarsysteem. De HUD zal dus gebruik moeten maken van een aantal tevoren gedefinieerde en meestal statische interfaces. Toch kunnen we ook hier onze objectgeoriënteerde ontwerpmethode toepassen.

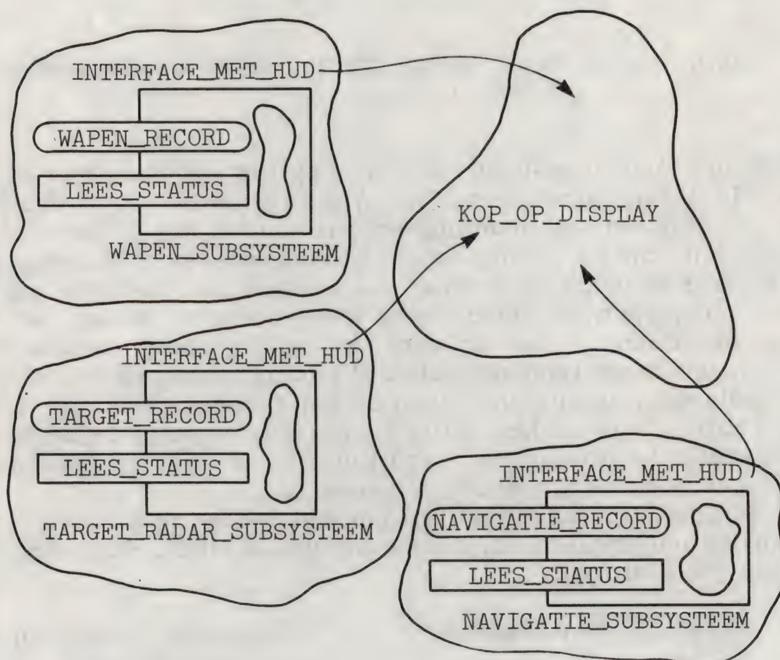
In figuur 21-2 is aangegeven hoe het systeem naar deelfuncties kan worden onderverdeeld, gezien vanuit de HUD. Het betreft de volgende subsystemen:

- WAPEN_SUBSYSTEMEN — besturen en richten van de boordwapens
- NAVIGATIE_SUBSYSTEEM — omvat alle apparatuur voor besturing en navigatie
- TARGET_RADAR_SUBSYSTEEM — zoekt en volgt het doel

Op het logische niveau ligt de structuur van onze oplossing in feite van tevoren vast; de objecten zijn de drie subsystemen en de HUD zelf. De wijze van implementatie van de objecten is nog niet van belang, maar wel moeten de interfaces tussen de objecten exact worden vastgelegd en dit kan gebeuren met behulp van Ada pakketbeschrijvingen. We veronderstellen dat we beschikken over de volgende externe objecten en bijbehorende operaties:

- WAPEN_SUBSYSTEEM.INTERFACE_MET_HUD
  object : WAPEN_RECORD
  operatie: LEES_STATUS
- NAVIGATIE_SUBSYSTEEM.INTERFACE_MET_HUD
  object : NAVIGATIE_RECORD
  operatie: LEES_STATUS
- TARGET_RADAR_SUBSYSTEEM.INTERFACE_MET_HUD
  object : TARGET_RECORD
  operatie: LEES_STATUS

De subsystemen bestaan zelf weer uit een groot aantal onderdelen, maar het is niet nodig van alle details van de subsystemen op de hoogte te zijn, om ze te kunnen gebruiken. Via het INTERFACE-_MET_HUD pakket worden alleen die grootheden geëxporteerd die



Figuur 21-3 Ontwerp voor de HUD interfaces.

voor de HUD van belang zijn. Via de componentselectie- of puntnotatie wordt alleen het INTERFACE_MET_HUD gedeelte uit de verschillende subsystemen benaderd en wordt meteen aangegeven dat de ontwikkelaars van de subsystemen verantwoordelijk zijn voor correcte interfaces met de HUD. Grafisch is dit nog eens in figuur 21-3 weergegeven.

De volgende stap is het maken van een formele beschrijving van de interfaces in Ada. We vullen in dit voorbeeld de details van de typerepresentaties niet in, maar bij een complete uitwerking van de oplossing zou dit nu moeten gebeuren. Als de objecten datarecords voorstellen, die vanuit één computersysteem naar een ander systeem moeten worden verzonden, kan Ada's representatiespecificatie worden gebruikt om de record lay-out zonodig tot op bitniveau vast te leggen. Ook hier zorgden we weer voor een modulair en vanuit de logische specificaties ontwikkeld ontwerp, in overeenstemming met de in hoofdstuk 4 ontwikkelde methodiek.

Binnen WAPEN_SUBSYSTEEM kan het interface als volgt worden geformuleerd:

```
package INTERFACE_MET_HUD is
  type WAPEN_RECORD is ...
  procedure LEES_STATUS(VAN_WAPEN : out WAPEN_RECORD);
end INTERFACE_MET_HUD;
```

De interface met NAVIGATIE_SUBSYSTEEM en TARGET_RADAR_SUBSYSTEEM ziet er net zo uit:

```
package INTERFACE_MET_HUD is
  type NAVIGATIE_RECORD is ...
  procedure LEES_STATUS(VAN_NAVIGATIE: out NAVIGATIE_RECORD);
end INTERFACE_MET_HUD;

package INTERFACE_MET_HUD is
  type TARGET_RECORD is ...
  procedure LEES_STATUS(VAN_TARGET : out TARGET_RECORD);
end INTERFACE_MET_HUD;
```

De ontwikkeling van de drie pakketbodies valt onder de verantwoordelijkheid van de respectievelijke subsysteem ontwikkelteams en hier hoeven wij ons dus niet bezig mee te houden. De TARGET_RADAR_SUBSYSTEEM groep zou bijvoorbeeld kunnen beslissen om hun procedure als een taakentry te implementeren, maar vanuit het abstractieniveau waarop wij ons nu bevinden is dit niet van belang.

De objecten waarmee we te maken krijgen hebben we nu beschreven. De volgende stap is het beschrijven van het HUD subsysteem zelf.

21.2 Ontwikkel Een Informele Strategie

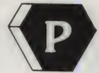


Voor de onderhoudbaarheid en begrijpelijkheid van het systeem is het van belang dat de software-oplossing een zo direct mogelijke afbeelding is van de werkelijkheid. Om tot een goede oplossing te kunnen komen kunnen we in dit geval het probleem het best vanuit het gezichtspunt van de piloot bekijken. Allereerst is van belang te weten welke objecten nu precies een rol spelen en daarbij kan de volgende informele strategie worden geformuleerd:

> **De piloot ziet een display dat bestaat uit vluchtparameters, een targetbox met een richtpunt, het target zelf met een toestandsvector en de wapen-status. Tijdens het aanvliegen van het doel veranderen deze elementen op willekeurige tijdstippen en de display wordt daarbij onmiddellijk aan-gepast. De werking van de HUD kan door de piloot via een commando worden beëindigd.**

We hebben natuurlijk heel wat details weggelaten, maar dat is niet zo erg. Zouden we in dit stadium al beginnen over de vergelijkingen voor de grootte van de target box of over de kleur van de display, dan zouden we door de bomen het bos niet meer zien. Dergelijke ontwerpbeslissingen kunnen vooralsnog worden uitgesteld en ook bijvoorbeeld hoe de piloot een doel bepaalt of zijn wapensysteem kiest, zijn vragen die binnen een ander subsysteem kunnen worden opgelost.

21.3 Formaliseer De Strategie



Als volgende stap gaan we de informele strategie formeel in Ada uitdrukken:

Identificeer de objecten en hun attributen

We onderstrepen weer de objecten uit de informele strategie, die bij de oplossing een rol zullen spelen:

> **De piloot ziet een <u>display</u> dat bestaat uit <u>vluchtparameters</u>, een <u>targetbox</u> met een <u>richtpunt</u>, het <u>target</u> zelf met een <u>toestandsvactor</u> en de <u>wapen-status</u>. Tijdens het aanvliegen van het doel veranderen deze elementen op willekeurige tijdstippen en de <u>display</u> wordt daarbij onmiddellijk aan-gepast. De werking van de <u>HUD</u> kan door de piloot via een <u>commando</u> worden beëindigd.**

Objecten zijn dus:

- TARGET met TOESTANDSVECTOR
- WAPENSTATUS
- VLUCHTPARAMETERS
- HEADS_UP_DISPLAY
- TARGET_BOX met RICHTfunctie
- GEBRUIKERSCOMMANDO

Het object HEADS_UP_DISPLAY ('Kop-op display') verwijst naar het fysieke beeldscherm en GEBRUIKERSCOMMANDO is louter bedoeld voor het beëindigen van het proces; de wijze waarop het proces wordt geactiveerd is afhankelijk van de gebruiksomgeving.

Identificeer de bewerkingen op de objecten

We onderstrepen de werkwoorden, die de operaties op onze objecten aangeven:

> **De piloot ziet een display dat bestaat uit vluchtparameters, een targetbox met een richtpunt, het target zelf met een toestandsvector en de wapenstatus. Tijdens het aanvliegen van het doel veranderen deze elementen op willekeurige tijdstippen en de display wordt daarbij onmiddellijk aangepast. De werking van de HUD kan door de piloot via een commando worden beëindigd.**

Operaties zijn:

- TARGET
  BEPAAL_WIJZIGINGEN
- WAPENSTATUS
  BEPAAL_WIJZIGINGEN
- VLUCHTPARAMETERS
  BEPAAL_WIJZIGINGEN
- HEADS_UP_DISPLAY
  PAS_AAN
- TARGET_BOX
  BEPAAL_WIJZIGINGEN
- GEBRUIKERSCOMMANDO
  IS_BEEINDIG

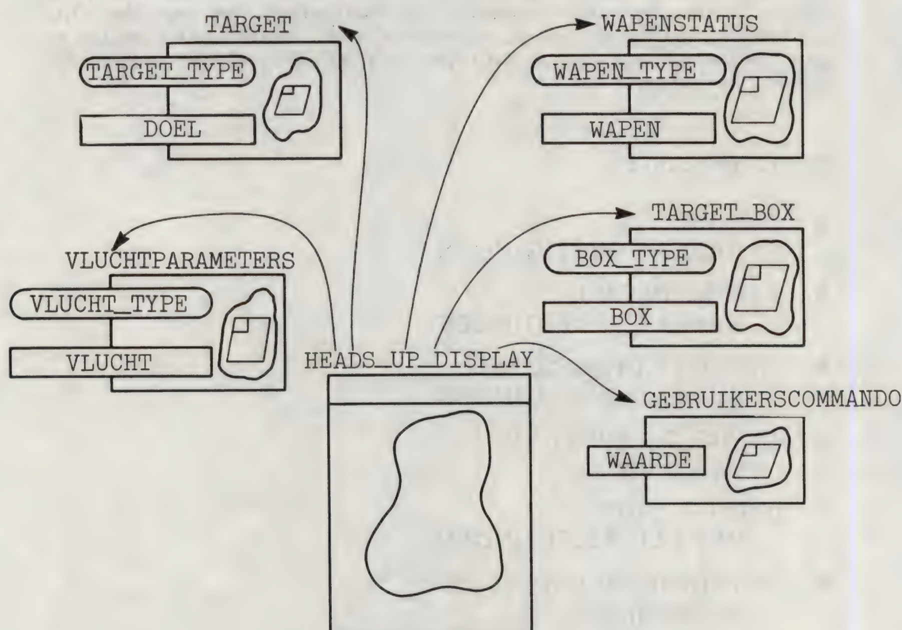
Bij het opstellen van deze lijst was het nodig goed op te letten op de gebruikte werkwoorden in de informele strategie. Er staat dat de verschillende objecten kunnen veranderen en niet dat wij ze zelf

kunnen veranderen; vandaar de operatie BEPAAL_WIJZIGINGEN. Verder is het mogelijk dat verschillende objecten tegelijkertijd een verandering ondergaan en daarom maken we van BEPAAL_WIJZIGINGEN een taakentry.

We hebben de operatie BEPAAL_WIJZIGINGEN met opzet 'overladen'; als we de oplossing verder uitwerken dan zal blijken dat dit de leesbaarheid van de oplossing ten goede komt. Ook de operatie PAS_AAN zullen we overladen, omdat immers alle objecten moeten kunnen worden aangepast. Wat betreft het GEBRUIKERSCOMMANDO kunnen we slechts vaststellen of dit IS_BEEINDIG (is).

Stel de interfaces vast

We zijn nu zo ver dat we de relaties tussen de verschillende objecten kunnen vaststellen. Figuur 21-4 geeft ons ontwerp nog eens schematisch weer. Op grond van de informele strategie gaat het om onderling onafhankelijke objecten (door taken voor te stellen), die elk een uniek gedeelte van de informatie omvatten (te beschrijven door middel van datatypen). Omdat hier elke taak onverbrekelijk met een bepaald datatype verbonden is, bedden we de objectdefinities in in pakketten. De objecten kunnen dan ook als bibliotheekeenheden worden gebruikt.



Figuur 21-4 Ontwerp voor HEADS_UP_DISPLAY.

Er gelden ook bepaalde zichtbaarheidseisen voor deze pakketten. Zo moet HEADS_UP_DISPLAY alle objecten kunnen zien, omdat elke wijziging in een object tot een wijziging via PAS_AAN van het display leidt. We namen ook de ontwerpbeslissing om de display in het hoofdprogramma te definiëren, omdat het immers het centrale element van de HUD is. De bodies van de PAS_AAN operaties werken we hier niet uit; hun implementatie wordt door subeenheden gerealiseerd. We zien hier dus een voorbeeld van top-down ontwerp (subeenheden) gecombineerd met bottom-up ontwerp (bibliotheekeenheden).

Misschien valt het u op dat de eerder gespecificeerde subsystemen in dit ontwerp niet meer worden vermeld. Dit is ook niet nodig, omdat deze eerder gespecificeerde interfaces op dit niveau niet van belang zijn. De nu in de oplossing gecreëerde objecten kunnen deze interfaces ieder direct zelf gebruiken. Het WAPENSTATUS object bijvoorbeeld, moet direct van de diensten van WAPEN_SUBSYSTEEM gebruik maken en VLUCHTPARAMETERS heeft direct NAVIGATIE-_SUBSYSTEEM nodig. Voor het hoogste abstractieniveau van de geformuleerde oplossing blijven dergelijke details echter verborgen.

We kunnen nu de details van de interfaces in het ontwerp gaan invullen en daartoe moeten we wat meer gegevens over onze objecten kennen, zoals bijvoorbeeld welke VLUCHTPARAMETERS er nu precies op het display verschijnen en hoe de TARGET_BOX er precies uitziet. Bij de verdere uitwerking van de oplossing zullen we deze vragen dan ook beantwoorden.

Voor TARGET kunnen we TARGET_TYPE met behulp van een STATUSVECTOR beschrijven. De STATUSVECTOR bevat gegevens over de positie en de snelheid van het doel. We zullen de ruimtelijke posities van de verschillende objecten steeds ten opzichte van hetzelfde coördinatenstelsel aangeven en we creëerden daartoe een pakket WERELDSYSTEEM, dat een aantal voor dit probleem benodigde typen bevat, waaronder de STATUSVECTOR. (Ook constanten als PI of ZWAARTEKRACHT_CONSTANTE kunnen in dit pakket worden opgenomen.)

Hier importeren we WERELDSYSTEEM:

```
with WERELDSYSTEEM;
package TARGET is
  --
  type TARGET_TYPE is new WERELDSYSTEEM.STATUSVECTOR;
  --
  task DOEL is
    entry BEPAAL_WIJZIGINGEN(DOEL: out TARGET_TYPE);
  end DOEL;
end TARGET;
```

De body van TARGET hebben we hier niet opgenomen, deze zal afzonderlijk worden uitgewerkt.

Het WAPENSTATUS pakket kan op een zelfde manier worden uitgewerkt. We geven aan hoe WAPEN_TYPE er uitziet en welke

informatie moet worden bijgehouden. We nemen de NAAM van het gegekozen wapen en het AANTAL dat beschikbaar is op:

```
package WAPENSTATUS is
  --
  type WAPEN is (VAST_BOORDWAPEN,RAKET,
                 DOELZOEKEND_BOORDWAPEN);
  subtype HOEVEELHEID is INTEGER range 0 .. 1_000;
  type WAPEN_TYPE is
    record
      NAAM    : WAPEN;
      AANTAL : HOEVEELHEID;
    end record;
  --
  task BEWAPENING is
    entry BEPAAL_WIJZIGINGEN(BEWAPENING: out WAPEN_TYPE);
  end BEWAPENING;
end WAPENSTATUS;
```

Als bovengrens voor HOEVEELHEID kozen we hier voor een numerieke waarde; verstandiger was misschien om als bovengrens een naam te importeren uit een globaal pakket, dat alle constanten bevat.

Vervolgens moeten we de structuur van VLUCHTPARAMETERS bepalen. Om het voorbeeld niet al te gecompliceerd te maken, nemen we alleen de HOOGTE en de AANVALSHOEK van het vliegtuig op. Later kunnen we alsnog beslissen grootheden zoals KOERS en WAARSCHUWING_BRANDSTOFNIVEAU op te nemen. Juist wegens de wijze waarop we ons ontwerp opbouwden, kunnen dergelijke wijzigingen eenvoudig worden aangebracht. Het ontwerp is modulair en veranderingen leiden alleen tot plaatselijke effecten. Het VLUCHT_PARAMETERS pakket wordt nu:

```
with WERELDSYSTEEM;
package VLUCHTPARAMETERS is
  --
  type VLUCHT_TYPE is
    record
      HOOGTE        : WERELDSYSTEEM.AFSTAND;
      AANVALSHOEK : WERELDSYSTEEM.RADIALEN;
    end record;
  --
  task VLUCHT is
    entry BEPAAL_WIJZIGINGEN(VLUCHT : out VLUCHT_TYPE);
  end VLUCHT;
end VLUCHTPARAMETERS;
```

We implementeren nu het interface voor TARGET_BOX. We omschreven eerder dat TARGET_BOX in grootte zou variëren en een cursor zou bevatten, die de RICHTing van het doelzoekend boordwapen bepaalt. In Ada kan dit pakket er als volgt uitzien:

```
with WERELDSYSTEEM;
package TARGET_BOX is
  --
  type RICHT_TYPE is
    record
      OP_TARGET : BOOLEAN;
      POSITIE   : WERELDSYSTEEM.COORDINAAT;
    end record;
  --
  type BOX_TYPE is
    record
      MIDDEN  : WERELDSYSTEEM.COORDINAAT;
      OMVANG  : WERELDSYSTEEM.DIMENSIE;
      RICHT   : RICHT_TYPE;
    end record;
  --
  task BOX is
    entry BEPAAL_WIJZIGINGEN(BOX : out BOX_TYPE);
  end BOX;
  --
end TARGET_BOX;
```

Tenslotte specificeren we GEBRUIKERSCOMMANDO:

```
package GEBRUIKERSCOMMANDO is
  task WAARDE is
    entry IS_BEEINDIG;
  end WAARDE;
end GEBRUIKERSCOMMANDO;
```

Voor GEBRUIKERSCOMMANDO was een taak eigenlijk voldoende geweest, maar omdat taken op zichzelf geen bibliotheekeenheid kunnen zijn, bedden we de taak in in een pakket.

De structuur van de operaties binnen HEADS_UP_DISPLAY zelf is iets anders dan die binnen de andere pakketten. Bedenk dat een hoofdprogramma zich gedraagt als een taak ten opzichte van zijn omgeving; we kunnen dus vanuit het hoofdprogramma entries van andere taken aanroepen. Voor het aanpassen van de display kunnen de subprogrammaspecificaties als volgt worden gedeclareerd:

```
procedure PAS_AAN_BEWAPENING(A : in WAPEN_TYPE);
procedure PAS_AAN_BOX       (B : in BOX_TYPE);
procedure PAS_AAN_VLUCHT    (V : in VLUCHT_TYPE);
procedure PAS_AAN_DOEL      (T : in TARGET_TYPE);
```

We gebruikten hier de typen die vanuit de andere pakketten worden geëxporteerd als subprogrammaparameters.

Nu we alle interfaces hebben beschreven, kunnen we de operaties op de objecten, zoals deze binnen de bodies worden beschreven, verder uitwerken.

Programmeer de operaties

De display moet alle objectspecificaties kunnen zien en krijgt daarom de volgende structuur:

```
with TARGET,WAPENSTATUS,VLUCHTPARAMETERS,
     TARGET_BOX, GEBRUIKERSCOMMANDO;
use  TARGET,WAPENSTATUS,VLUCHTPARAMETERS,
     TARGET_BOX,GEBRUIKERSCOMMANDO;
procedure HEADS_UP_DISPLAY is
  WAPEN_DATA   : WAPEN_TYPE;
  BOX_DATA     : BOX_TYPE;
  VLUCHT_DATA  : VLUCHT_TYPE;
  TARGET_DATA  : TARGET_TYPE;
  procedure PAS_AAN_BEWAPENING(A: in WAPEN_TYPE)  is separate;
  procedure PAS_AAN_BOX        (B: in BOX_TYPE)    is separate;
  procedure PAS_AAN_VLUCHT     (V: in VLUCHT_TYPE) is separate;
  procedure PAS_AAN_DOEL       (T: in TARGET_TYPE) is separate;
begin
  -- body van hoofdprogramma
end HEADS_UP_DISPLAY;
```

Omdat de PAS_AAN bodies als subeenheden worden gedeclareerd kunnen ze afzonderlijk worden gecreëerd en gecompileerd. Dat is vooral gemakkelijk in de testfase: we kunnen eerst 'stubs', dat wil zeggen lege en later in te vullen 'dummies' als bodies gebruiken en stap voor stap kunnen de verschillende functies worden toegevoegd.

Nu moeten nog de bodies van de overige programma-eenheden worden uitgewerkt. De volledige oplossing is echter dermate gecompliceerd, dat we hem niet in dit voorbeeld kunnen uitwerken. Wel hebben we genoeg informatie om een ander onderdeel van het ontwerp te kunnen uitwerken: de body van HEADS_UP_DISPLAY zelf.

Deze taak moet met behulp van BEPAAL_WIJZIGINGEN de veranderingen in de andere objecten, die op willekeurige momenten kunnen plaatsvinden, kunnen signaleren. Met behulp van Ada's taakstructuur is dat vrij gemakkelijk te realiseren. Op het eerste gezicht zou men misschien denken dat het werken met taken een nogal omslachtige oplossing is; waarom geen directe subprogramma-aanroepen? Het antwoord luidt als volgt: door de display als een hoofdprogrammataak te behandelen, kunnen de andere objecten een rendez-vous uitvoeren met het hoofdprogramma, de benodigde gegevens uitwisselen en onmiddellijk met hun eigen werk verder gaan, zonder te hoeven wachten totdat de display klaar is met het verwerken van andere wijzigingen binnen andere objecten. Ook de randvoorwaarden ten aanzien van de verwerkingstijden kunnen door gebruik te maken van taken op een flexibele wijze worden opgenomen. Op grond van het vastgestelde scenario gedraagt de display zich als een berichtenverzendend systeem (zie hoofdstuk 16):

```
begin
  loop
    select
      DOEL.BEPAAL_WIJZIGINGEN(TARGET_DATA);
      PAS_AAN_TARGET(TARGET_DATA);
    else
      null;
    end select;
    select
      BEWAPENING.BEPAAL_WIJZIGINGEN(WAPEN_DATA);
      PAS_AAN_BEWAPENING(WAPEN_DATA);
    else
      null;
    end select;
    select
      VLUCHT.BEPAAL_WIJZIGINGEN(VLUCHT_DATA);
      PAS_AAN_VLUCHT(VLUCHT_DATA);
    else
      null;
    end select;
    select
      GEBRUIKERSCOMMANDO.WAARDE.IS_BEEINDIG;
      exit;
    else
      null;
    end select;
  end loop;
end HEADS_UP_DISPLAY;
```

Er zit opvallend veel regelmaat in bovenstaande instructies door het gebruik van een aantal voorwaardelijke entry-aanroepen. Het HEADS_UP_DISPLAY loopt één voor één de andere objecten af ('polling') om na te gaan of er nieuwe gegevens beschikbaar zijn. Als een rendez-vous meteen mogelijk is, dan worden de gegevens aan lokale variabelen doorgegeven en wordt de bedienende taak vrijgegeven om zijn werk te kunnen voortzetten. De HEADS_UP-_DISPLAY roept inmiddels de PAS_AAN operatie aan. Is een rendez-vous niet onmiddellijk mogelijk dan wordt alleen de entry voor de volgende taak getest en gebeurt er verder niets. De lus wordt verlaten als de entry IS_BEEINDIG binnen GEBRUIKERSCOMMANDO wordt geaccepteerd. We gaan daarbij uit van de veronderstelling, dat ook alle andere taken een **terminate** mogelijkheid bevatten om het totale systeem ordelijk af te sluiten.

De display zou gemakkelijk zo kunnen worden aangepast dat ook andere objecten kunnen worden behandeld, of dat aan bepaalde randvoorwaarden ten aanzien van verwerkingstijdstippen is voldaan. Door bijvoorbeeld van een **delay**-instructie gebruik te maken zou de programmalus met vaste tussentijden (bijvoorbeeld om de 100 seconden) kunnen worden herhaald. Zouden we daarbij binnen een bepaalde taak gegevens met andere tussenintervallen willen inlezen,

dan is ook dat mogelijk door een voorwaardelijke entry te herhalen (om de meetfrequentie te verhogen), of door een voorwaarde aan de select-instructie te verbinden, zodat die bijvoorbeeld maar om de andere keer wordt uitgevoerd (als we de meetfrequentie willen verlagen). Dergelijke verfraaiingen laten we als oefening aan de lezer over.

Desgewenst zijn er ook mogelijkheden om de betrouwbaarheid tijdens de werking van het systeem nog te verhogen; we kunnen bijvoorbeeld een else-clausule toevoegen:

```
select
  BOX.BEPAAL_WIJZIGINGEN(BOX_DATA);
else
  TEST_HUD;
end select;
```

De procedure TEST_HUD zou daarbij een automatische test van het systeem op correcte werking kunnen uitvoeren. Telkens wanneer er wordt geconstateerd, dat zich geen veranderingen voordeden wordt deze test uitgevoerd.

Hierbij laten we het wat betreft de uitwerking van dit systeem. Bij verdere uitwerking van de pakketbodies zouden erg veel details moeten worden betrokken, hoewel de totale structuur vrij regelmatig blijft. Een directe afbeelding van het probleemgebied naar een oplossingsruimte hebben we nu gecreëerd en verdere verfijning kan gebeuren met behulp van dezelfde objectgeoriënteerde ontwerptechnieken die we in het voorgaande toepasten. Stapsgewijs kunnen we zo het totale systeem ontwikkelen, waarbij het steeds mogelijk blijft de onderdelen afzonderlijk te testen. Weliswaar zou de totale oplossing nogal veel programmatekst omvatten, maar toch zijn we op deze manier steeds in staat de complexiteit van het geheel te blijven beheersen.

Oefeningen

1. Voeg aan de specificatie van VLUCHTPARAMETERS gegevens voor KOERS en BRANDSTOFNIVEAU toe.

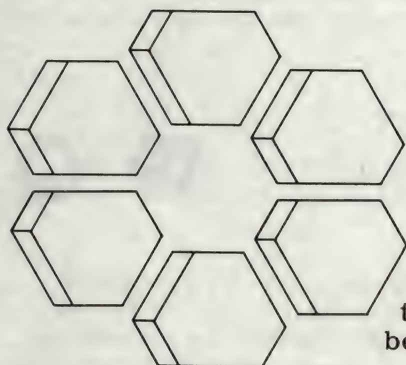
*2. Pas de body van HEADS_UP_DISPLAY zo aan, dat de instructies om de 100 seconden worden herhaald.

*3. Pas HEADS_UP_DISPLAY verder aan, zodat WAPENSTATUS maar tijdens elke derde cyclus wordt ingelezen.

*4. Pas HEADS_UP_DISPLAY aan, zodat TARGET tweemaal binnen elke cyclus wordt ingelezen.

Pakket 8

PROGRAMMEREN MET ADA



Een programmeur zou geen programmeur zijn, als hij zijn programma af en toe niet louter als esthetisch object zou beschouwen.

G.M. Weinberg
The Psychology of Computer Programming [1]

22 DE ADA PROGRAMMEEROMGEVING

Algemeen gesteld is een programmeeromgeving het gebied waarbinnen softwaresystemen worden ontwikkeld. Voor de programmeur bevat dit gebied softwaregereedschappen, ontwikkelmethodes, talen en andere programmeurs. De mate van volledigheid en verfijndheid van een programmeeromgeving heeft meer invloed op de kwaliteit van de ontwikkelde software dan welke programmeertaal ook. In hoofdstuk 3 beschreven we hoe het Department of Defense dit inzag en de ontwikkeling van specificaties begeleidde voor een software-instrumentarium ter ondersteuning van toepassingen in Ada. Het betreft hier de Ada programmeeromgeving (Ada Programming Support Environment of APSE), waarvan we in dit hoofdstuk de mogelijkheden zullen beschrijven. We zouden over dit onderwerp gemakkelijk een heel boekdeel kunnen vullen, maar hier zullen we ons beperken tot de hoofdzaken.

22.1 Enige Opmerkingen over Programmeeromgevingen In Het Algemeen





Als we de bestaande programmeeromgevingen aan een onderzoek onderwerpen, dan blijkt dat de meeste maar een beperkte hoeveelheid programmeergereedschappen bevatten, waaronder compilers, programmabibliotheken, passieve editors, linkers (dit is programmatuur om afzonderlijke programma-eenheden te koppelen) en loaders (programmatuur om de programma-eenheden in het geheugen te laden). Verder komt wel testprogrammatuur voor en meestal programmatuur voor bestandsbeheer. Dat we niet over een uitgebreider instrumentarium beschikken is onder andere te wijten aan het grote aantal verschillende programmeertalen dat zich sinds het einde van de jaren zestig heeft verbreid (zie nog eens hoofdstuk 2). Binnen een computersysteem dat meerdere programmeertalen ondersteunt is meestal wel een groot aantal programmeergereedschappen voorhanden, maar zelden vormen deze een totale set, waarbinnen de onderdelen goed op elkaar aansluiten en geschikt zijn voor één bepaalde taal.

Het probleem wordt nog ingewikkelder als we er het grote aantal computersystemen bij betrekken. De programmeergereedschappen zijn vaak sterk machine-afhankelijk en het is daarom moeilijk en

soms zelfs onmogelijk om deze gereedschappen van de ene naar de andere computer over te zetten.

De gereedschapsset is zelden compleet; het uitschrijven van de programmatekst ('coderen') is maar een klein onderdeel van de totale software levenscyclus; toch bestaan er voor het coderen de meeste hulpmiddelen. Projectleiders bij de ontwikkeling van softwaresystemen hebben behoefte aan hulpmiddelen voor het realiseren van projectplanningen, het toewijzen van de middelen en het bewaken van de projectvoortgang. Binnen de meeste omgevingen is er geen gereedschapsset voorhanden, die al deze onderdelen integraal ondersteunt.

Deze situatie remt de produktiviteit tijdens de ontwikkeling van het systeem. Softwareprodukten zijn uiteindelijk bedoeld voor een computer, maar tot nu toe heeft men weinig succes geboekt bij het gebruiken van de computer bij de beheersing van de software levenscyclus.

22.2 Achtergrond Van De Ada Programmeeromgeving



Al bij het begin van het DoD gemeenschappelijke hogere programmeertaalproject zag Whitaker in, dat het definiëren van een programmeertaal een noodzakelijke, maar geen voldoende activiteit was ter bestrijding van de problemen van de softwarecrisis. DoD ondersteunde daarom de ontwikkeling van specificaties voor een programmeeromgeving ter aanvulling van de mogelijkheden van Ada zelf. Deze specificaties werden geformuleerd in STONEMAN. In het STONEMAN rapport wordt vermeld, dat het doel van APSE is: "ondersteuning van ontwikkeling en onderhoud van toepassingen in Ada tijdens de gehele levenscyclus, met bijzondere nadruk op software voor ingebedde computertoepassingen" [1,2].

Het doel van STONEMAN is, een aantal problemen die wij in de vorige paragraaf signaleerden, te elimineren. Uiteindelijk wordt Ada de gemeenschappelijke taal voor alle 'embedded' toepassingen van het DoD en de ontwikkeling van een standaard APSE is daarbij van groot belang en kan in het bijzonder tot de volgende resultaten leiden [3]:

- Lagere compiler-ontwikkelingskosten
- Lagere ontwikkelingskosten voor hulpgereedschap
- Betere software-overdraagbaarheid
- Verbeterde algemene inzetbaarheid van programmeurs

In de volgende paragraaf gaan we nader in op de STONEMAN specificaties voor APSE.

22.3 Architectuur Van De Ada Programmeeromgeving



Een van de basisideeën van APSE is de gast-/doelcomputer filosofie. De STONEMAN specificaties gaan uit van de gedachte dat APSE wordt gebruikt op een gastcomputer voor de ontwikkeling van een systeem voor een bepaalde doelcomputer. (De doelcomputer kan eventueel dezelfde zijn als de gastcomputer.) Vooral bij de ontwikkeling van ingebedde systemen zijn gast- en doelcomputer meestal niet gelijk: de doelcomputer is meestal veel kleiner en kan niet het instrumentarium bevatten dat nodig is om het systeem te ontwikkelen.

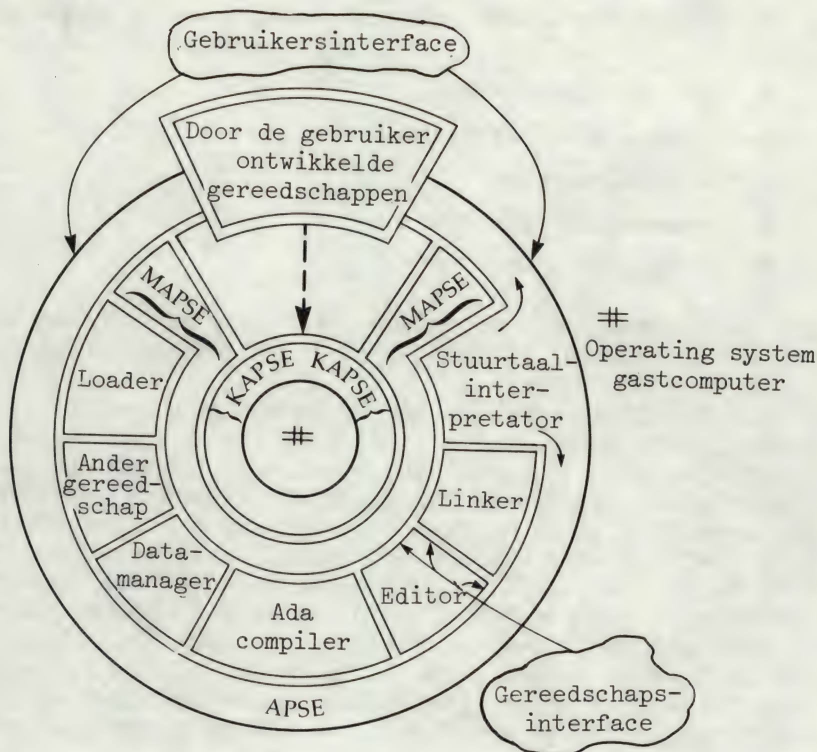
Een ander belangrijk onderdeel van APSE is de programma database. In hoofdstuk 20 zagen we al, dat Ada kan werken met een programmabibliotheek van afzonderlijk gecompileerde programma-eenheden. APSE voegt hier een database aan toe, waarin alle projectinformatie kan worden opgenomen, zoals de source- en objectcodes van programma's, de programmadocumentatie en andere gegevens. Zo kan de projectbeheerder beschikken over een centraal gegevensbestand ter controle van de projectvoortgang. Ook bevat APSE, zoals we nog zullen zien, hulpmiddelen voor het zogenaamde configuration management: het beheer van verschillende versies van het ontwikkelde systeem. Hiervan kan gebruik worden gemaakt bij het vaststellen van mijlpalen of 'baselines', waaraan de projectvoortgang kan worden getoetst.

STONEMAN vereist verder dat APSE uitbreidbaar moet zijn en dat alle gereedschappen in Ada moeten zijn geprogrammeerd. Over het gebruik van een stuurtaal binnen APSE spreekt STONEMAN niet, maar het ligt voor de hand dat Ada zelf daarvoor kan worden aangewend.

STONEMAN onderscheidt een aantal niveaus in de Ada programmeeromgeving, zoals schematisch aangegeven in figuur 22-1 [4]. De kern wordt gevormd door het operating system van de gastcomputer. Daaromheen ligt de zogenaamde *Kernel Ada Programming Support Environment* of KAPSE, waarin de vertaling van de logische gereedschappen uit APSE naar het fysieke niveau van de computer plaatsvindt. Boven KAPSE ligt MAPSE, de *Minimal Ada Programming Support Environment* met een minimaal benodigde set gereedschappen. Het hoogste niveau wordt gevormd door APSE met een meer verfijnd instrumentarium ter ondersteuning van de hele software levenscyclus.

KAPSE

KAPSE bevat alleen datgene wat nodig is voor ondersteuning van de verdere omgeving tijdens de verwerking; KAPSE legt de relatie vast tussen APSE en de fysieke machine. KAPSE zorgt voor de systeemoverdraagbaarheid; willen we dezelfde programmeeromgeving op een andere computer overzetten, dan hoeft voor deze nieuwe computer alleen maar een nieuwe KAPSE te worden geschreven.



Figuur 22-1 De Ada programmeeromgeving APSE.
Uit: Wolf, Babich, Simpson, Tholl en Weissman, "The Ada Language System", *Computer* 1981

MAPSE

In MAPSE zit een minimale gereedschapsset van softwaregereedschappen. Dit is het eerste niveau dat voor de programmeur zichtbaar is en dat alle basisgereedschappen bevat, nodig bij het ontwikkelen van programma's. Volgens de STONEMAN specificaties bevat MAPSE de volgende gereedschappen:

- text editor
- 'pretty printer' (een programma om programma's met een gestructureerde lay-out af te drukken)
- compiler
- linker
- set-use static analyzer

- control-flow static analyzer (analyse van gevolgde instructiepaden)
- dynamic analysis tools
- terminal interface routines
- file administrator
- command interpretator
- configuration manager

Elke implementatie kan gereedschap aan deze set toevoegen of gereedschap weglaten; STONEMAN geeft slechts suggesties ten aanzien van op te nemen tools.

STONEMAN eist wel, dat de verschillende gereedschappen een gecoördineerd geheel vormen, zodat het gemakkelijk is van het ene gereedschap op het andere over te stappen. Om software tussen gereedschappen gemakkelijk overdraagbaar te maken wordt vaak van een 'tussentaal' gebruik gemaakt, waarnaar de sourcecode eerst wordt vertaald. Deze tussentaal: *Descriptive Intermediate Attributed Notation for Ada* of DIANA, is meestal de output van de eerste compilatieslag en deze vertaling bevat alle syntactische en semantische informatie bij een bepaalde programma-eenheid. De DIANA vertaling kan gemakkelijk worden gebruikt om de software op een andere computer over te zetten of er kunnen gereedschappen op worden losgelaten, die het programma op het niveau van source- of objectcode analyseren.

APSE

APSE is de totale programmeeromgeving en bevat meer en verfijndere gereedschappen dan MAPSE. STONEMAN vereist dat er gereedschappen in APSE ter beschikking zijn voor de volgende toepassingen:

- het creëren van database objecten
- het aanbrengen van wijzigingen
- het uitvoeren van analyses
- het transformeren van gegevens en/of programmatuur
- display van gegevens
- uitvoeren van programma's
- onderhoud

Dit niveau wordt door STONEMAN het minst in detail gespecificeerd, maar biedt juist de meeste mogelijkheden voor daadwerkelijke vernieuwing. In APSE kunnen krachtige instrumenten zijn opgenomen als syntaxgerichte editors, real-time debuggers en misschien zelfs hulpmiddelen voor automatisch genereren van programma's.

Op dit hoogste niveau zijn er twee categorieën van gereedschappen te onderscheiden:

- *Generieke gereedschappen*
  Dit zijn gereedschappen die op alle mogelijke programma's kunnen worden toegepast, onafhankelijk van het toepassingsgebied. Denk aan compilers, linkers, loaders en dergelijke.
- *Methode-afhankelijke gereedschappen*
  Hier gaat het om gereedschap voor een bepaald toepassingsgebied. In deze categorie vallen preprocessors en configuration managers.

Het Ada Joint Program Office ondersteunde de ontwikkeling van functionele gereedschappen in beide categorieën en streeft naar een discipline van programmeren, die niet door de een of andere willekeurige verzameling van gereedschap wordt bepaald.

Duidelijk is dat een APSE niet onontbeerlijk is om toch Ada programmatuur te kunnen ontwikkelen: er zijn al enkele Ada implementaties, die onder standaard operating systemen draaien. Ada in combinatie met een goede APSE biedt echter wel een krachtig hulpmiddel voor het ontwikkelen en onderhouden van softwareprodukten. In het volgende hoofdstuk gaan we in op de software levenscyclus en bekijken we hoe Ada en APSE ons tijdens de hele cyclus tot steun kunnen zijn.

Oefeningen

*1. Vergelijk de UNIX omgeving met APSE.

2. Hoe past het APSE bibliotheekmodel binnen het Ada compilatiemodel, zoals behandeld in hoofdstuk 20?

23 ADA EN DE SOFTWARE LEVENSCYCLUS

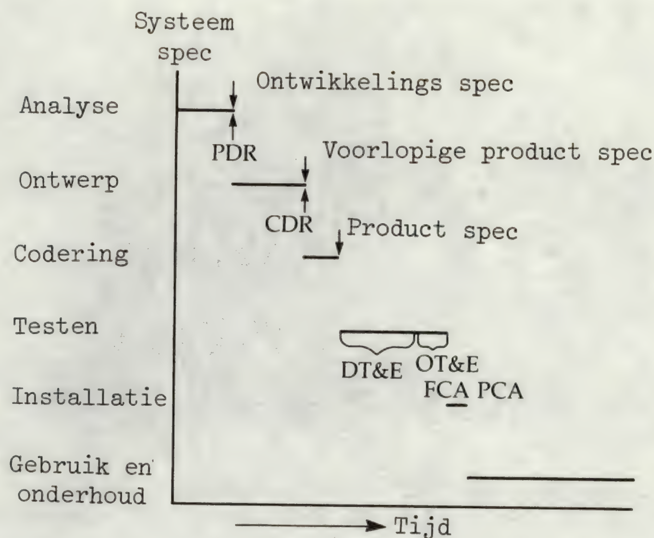
De levenscyclus van software begint bij het allereerste idee en strekt zich uit tot het uiteindelijke gebruik en het onderhoud van de programmatuur. In het verleden werden de verschillende fasen in de levenscyclus door software ontwikkelaars vaak behandeld alsof zij volledig onderling onafhankelijk waren. Het systeem werd dan ontworpen met de ene techniek (bijvoorbeeld met gebruik van pseudocode), gecodeerd in een andere daarmee niet in verband staande taal en vervolgens getest met behulp van gereedschappen die weer volstrekt niets met de eerdere fasen uitstaande hadden. Deze benadering leidde tot tal van problemen en in ieder geval tot nachtmerries voor de configuratiemanager en tot modules die niet in onderlinge samenwerking functioneerden. Uiteindelijk werd het systeem wel opgeleverd, maar niet zonder veel geweeklaag en tandengeknars!

Binnen de software levenscyclus zijn zes belangrijke fasen te onderscheiden. Ze krijgen in de literatuur vaak verschillende benamingen, maar hier zullen wij ze zo beschrijven:

- analyse
- vaststellen programma van eisen
- ontwerp
- codering
- testen
- in gebruikname en onderhoud

In figuur 23-1 zijn deze fasen in de tijd weergegeven en daarbij hebben we enkele specifieke te produceren documenten genoemd en een aantal subfasen.

Grote systemen ontstaan niet spontaan; zij ontwikkelen zich uit kleinere. Het ideaal is een ontwikkeling met consistente gereedschappen en notaties: we moeten onze programmeertaal niet allen gebruiken voor het coderen maar ook voor de formulering van het ontwerp. Ada biedt een stuk gereedschap dat in deze zin gedurende de gehele levenscyclus kan worden toegepast. Ada is een ontwerptaal (zie hoofdstuk 6); hier zullen we onderzoeken wat dit precies inhoudt.



Figuur 23-1 De software levenscyclus.

23.1 De Fase Van De Probleemanalyse



De eerste fase in de levenscyclus is die van de probleemanalyse. Allereerst moet een goed inzicht in de probleemstelling worden verkregen en moet worden nagegaan in hoeverre automatisering bij de oplossing een hulpmiddel kan zijn. Als besloten is tot automatisering over te gaan, worden vervolgens de functies van de te ontwikkelen software beschreven in een systeemspecificatie. Ook wordt alvast onderzocht welke apparatuur kan worden gebruikt, welke opslagcapaciteit deze dient te hebben en in hoeverre nieuw personeel moet worden aagetrokken.

In dit stadium is de vraag of bij de oplossing al of niet van Ada gebruik zal worden gemaakt nog volstrekt niet relevant. We zijn bezig met de probleemstelling en het gebruik van Ada is een onderdeel van de oplossing en daaraan zijn we nog niet toe. Bij de probleemanalyse behoort de te gebruiken programmeertaal niet te worden betrokken; Ada komt pas later aan bod. Iets anders is het als we een geschikte APSE tot onze beschikking hebben. In dat geval kan van deze programmeeromgeving worden gebruik gemaakt om met behulp van documentatieprogrammatuur de systeemspecificatie op te stellen. In een later stadium van de ontwikkeling kan dan gecontroleerd worden of de oplossing met deze specificatie overeenkomt.



Ada als kind. Gravure door W.H. Mote, naar een tekening van F. Stone (Murray collectie).

23.2 Vaststellen Van Het Programma Van Eisen



Deze fase (ook wel de fase van het functioneel ontwerp genoemd) begint met goedkeuring van de globale systeembeschrijving en wordt afgesloten met het voltooien van een rapport dat het (functioneel) ontwerp bevat, in de Angelsaksische literatuur Preliminary Design Review genaamd. In deze fase worden de functies, zoals vastgesteld tijdens de probleemanalyse, nader uitgewerkt en wordt in detail vastgesteld welke functies de te ontwikkelen software zal moeten bevatten. De verleiding is groot in dit stadium om al complete oplossingen te gaan uitwerken, maar de bedoeling is op een hoger abatractieniveau te blijven en alleen de functies te beschrijven. Deze beschrijving behoort dan uit te monden in een door alle partijen goedgekeurd functioneel ontwerprapport.

We gaan nu uit van de veronderstelling, dat Ada als ontwerp- en specificatietaal wordt gebruikt. Net zoals bij de vijf behandelde ontwerpproblemen kan de Ada programma-eenheid worden gebruikt ter specificatie van dit ontwerp op logisch niveau. De ontwerper dient daarbij de oplossing te beschouwen vanuit een beschrijvend

gezichtspunt en niet te vervallen in de gebiedende (imperatieve) vorm, waartoe men door zoveel programmeertalen in een veel te vroeg stadium gedwongen wordt. Zo is het mogelijk, zoals Wheeler stelt, dat "de beperkingen die aan de systeemstructuur worden opgelegd door het gebruik van Ada als documentatietaal bij het systeemontwerp, er niet alleen voor zorgen dat het ontwerpen en de implementatie worden vergemakkelijkt, maar ook dat het uiteindelijke systeem onderhoudbaarder wordt." [1].

De levenscyclus kan nu op een geïntegreerde manier worden benaderd; we beschikken in Ada over een uitdrukkingsmiddel voor elke fase. Ook hier geldt: als we binnen een goede APSE kunnen werken, dan kunnen we van configuration management hulpmiddelen gebruik maken om de verschillende versies en de fasen in het ontwerp te administreren en te documenteren. De APSE kan verder ook worden gebruikt voor het continu controleren in hoeverre de oplossingen voldoen aan de specificaties.



Ada op negentienjarige leeftijd.

23.3 De Ontwerpfase



Deze fase (ook wel de fase van het technisch ontwerp genoemd) begint als het *Preliminary Design Report* (PDR) is opgeleverd en eindigt als het zogenaamde *Critical Design Report* (CDR) gereed is. (In Nederland wordt wel gesproken van Functioneel Systeem Ontwerp (FSO) en Technisch Systeem Ontwerp (TSO).) In deze fase worden de functionele specificaties verder uitgewerkt en worden de oplossingen geformuleerd, waaronder exacte beschrijvingen van de relaties (interfaces) tussen de onderscheiden eenheden en gedetailleerde procesbeschrijvingen.

Bij deze verdere uitwerking blijkt vaak, dat de aanstaande gebruiker niet alle implicaties van de eerder specificaties heeft (kunnen) overzien en de ontwerpfase is daarom vaak een iteratief proces. Er worden wijzigingen in de specificaties aangebracht en die leiden weer tot wijzigingen in het ontwerp. Dergelijke wijzigingen zijn op zichzelf (als ze tenminste niet al te groot zijn) een goed teken: in ieder geval is er sprake van communicatie tussen opdrachtgever en ontwerper. Voor de manager is wellicht het beste hulpmiddel ter bewaking van het ontwerpproces de 'structured walkthrough', het op systematische wijze doorlopen van het ontwerp. Indien dit goed gebeurt zal het leiden tot een qua structuur en functies begrijpelijker produkt.

In de ontwerpfase kan Ada direct worden ingezet. Voordat Ada bestond zouden we ons ontwerp hebben gedocumenteerd met behulp van stroomdiagrammen of met een 'Process Design Language' (PDL). Nu kan Ada zelf als PDL worden gebruikt en is de uiteindelijke oplossing een uitwerking van de beschrijving in de PDL. Er zijn twee benaderingen van het gebruik van Ada als PDL. De eerste benadering, ontwikkeld door de IBM Federal Systems Division, gebruikt een 'ontwerptaal' subset van Ada als PDL, met daarbij uitgebreide documentatie [2]. Als tweede benaderingswijze stelt H. Hart het gebruik voor van een vrijere vorm van Ada als PDL, onder weglating van een aantal voor Ada geldende syntaxregels [3].

De IBM benadering legt de nadruk op hiërarchische onderverdeling, definitie van interfaces en modulariteit, maar men moet tijdens het gebruik rekening houden met de syntaxregels en de semantiek van Ada. Een grondige kennis van Ada is dus noodzakelijk voordat men met deze PDL met succes kan werken. Bij de benadering van Hart is veel minder aandacht voor de Ada syntax nodig en kan dus eerder succes worden geboekt. Beide methoden kunnen naar onze mening worden toegepast, maar het gebruik van een syntactisch correcte subset van Ada verdient toch de voorkeur.

Als we er zorg voor dragen dat onze programma-eenheden en objecten zinvolle namen krijgen, dan wordt het ontwerp in Ada vrijwel 'zelfdocumenterend' bij zelfs middelgrote softwaresystemen (zie Appendix B). Begeven we ons binnen het gebied van de grote tot zeer grote systemen, dan is interne documentatie van de genomen ontwerpbeslissingen van het allergrootste belang. Welke ontwerp-



Ada, gravin van Lovelace, op ongeveer zevenentwintigjarige leeftijd, naar een tekening van A.E. Chalon (Murray collectie).

benadering ook wordt gekozen, wij blijven de nadruk leggen op een beschrijvende (declaratieve) benadering van de oplossing, zolang men zich op het hoogste logische abstractieniveau bevindt. De objectgeoriënteerde ontwerpmethode kan daarbij als ondersteuning dienen. Op de lagere abstractieniveaus, wanneer de functies nader moeten worden gedefiniëerd, zijn de top-down decompositietechnieken voldoende krachtig.

Zoals we bij de uitwerking van onze ontwerpproblemen lieten zien, kan het ontwerp worden geformuleerd door gebruik te maken van de Ada specificaties voor programma-eenheden, om op die manier de interfaces tussen de objecten in de oplossingsruimte formeel te beschrijven. Voor de bodies van de eenheden kunnen zolang 'stubs' of 'stoplappen' worden gebruikt, waaraan commentaar ten aanzien van de uiteindelijk te implementeren functie wordt toegevoegd. Het voordeel van deze werkwijze is, dat het nu mogelijk is

het systeem al in een vroegtijdig stadium van het ontwerp te compileren en op deze wijze eventuele logische inconsistenties te ontdekken (uitgaande van de IBM-benadering voor de PDL). Als binnen de APSE geschikt gereedschap aanwezig is kan ook al configuration management worden uitgevoerd en kunnen versies van gereedgekomen eenheden worden geadministreerd. Verder kunnen we de specificaties blijven controleren en de APSE gebruiken voor de overige externe documentatie.

23.4 De Fase Van Het Coderen



De fase van het coderen begint als het *Critical design Review* (CDR) gereed is, maar wanneer deze fase eindigt is moeilijker te zeggen. Eigenlijk wordt de coderingsfase pas afgesloten als we de software aan zijn lot overlaten en niet meer onderhouden. Het belangrijkste document voor deze fase is de volledige produktspecificatie met een beschrijving van de software, zoals deze geïmplementeerd is.

Als Ada als implementatietaal wordt gekozen, dan is deze fase betrekkelijk triviaal en in feite slechts een voortzetting van het proces dat in de ontwerpfase werd begonnen. We behoren op dit punt te beschikken over een gedetailleerd ontwerp, uitgedrukt in een aantal Ada programma-eenheden met volledige specificaties. Tijdens de fase van het coderen hoeven alleen de bodies nog maar te worden uitgewerkt. Natuurlijk zal het daarbij geregeld voorkomen dat we ontdekken dat een verdere onderverdeling in programma-eenheden noodzakelijk is. De ontwerp- en coderingsfase lopen nu in feite in elkaar over; of beter: er is sprake van een iteratief proces tussen ontwerp en implementatie.

Als het uiteindelijke programma niet in Ada wordt geschreven, bijvoorbeeld omdat de doelcomputer Ada niet ondersteunt, dan kan toch van het Ada ontwerp gebruik worden gemaakt en kan dit naar de op de doelcomputer gebruikte taal worden geconverteerd. Het op een systematische wijze vertalen van een gestructureerd ontwerp in Ada naar bijvoorbeeld een assembleertaal met veel minder structuur zal echter minder problemen opleveren dan direct coderen in assembler.

Een schema met behulp waarvan ons Ada ontwerp kan worden vertaald naar een andere hogere programmeertaal, zoals FORTRAN of JOVIAL zou eveneens kunnen worden ontwikkeld. Naar schatting zou 75% van de conversie automatisch kunnen gebeuren (afhankelijk van de aard van de toepassing). De APSE kan weer worden gebruikt voor configuration management en voor documentatiedoeleinden. Verfijnde APSE gereedschappen, zoals syntaxis georiënteerde editors kunnen het werk nog vergemakkelijken, omdat zij ervoor kunnen zorgen dat alleen syntactisch correcte programma-eenheden worden geproduceerd.



Ada omstreeks 1850. (In bezit van Mrs. Doris Langley Moore.)

23.5 De Fase Van Het Testen



In de testfase wordt de werking van de software gecontroleerd en vergeleken met de specificaties. Men hoeft met het ingaan van deze fase niet te wachten totdat alle programma's gereed zijn: de 'ontwerp een stukje, codeer een stukje en test een stukje' benadering is veel beter. Fouten worden zo veel eerder ontdekt en kunnen eerder worden gecorrigeerd.

Bij militaire toepassingen wordt de testfase vaak in twee gedeelten onderverdeeld, in het Engels *Development Test and Evaluation* (DT&E) en *Operational Test and Evaluation* (OT&E) genoemd. De eerste test betreft de afzonderlijke modulen en de tweede de integratietest van het totale systeem in de gebruiksomgeving. De DT&E test beslaat, naarmate meer modulen gereedkomen, een steeds groter

deel van het totale systeem. Zijn deze tests voltooid, dan wordt de AT&E uitgevoerd: een totaaltest binnen een omgeving die zoveel mogelijk overeenkomt met de gebruiksomgeving van het systeem. Vaak betekent dit dat het systeem parallel draait met een bestaand systeem in de operationele omgeving. De belangrijkste documenten uit deze fase zijn testrapporten en testcertificaten.

Tijdens de testfase kan gebruik worden gemaakt van APSE gereedschap zoals een symbolische debugger. Het moelijkste in deze fase is om een standaardversie (baseline versie) van het systeem in stand te houden. Op grond van de uitgevoerde tests en de daarbij gevonden fouten moeten kleine veranderingen worden aangebracht in afzonderlijke programma-eenheden en vooral bij grote systemen wordt het dan extra moeilijk een totaaloverzicht te houden. Ook hier kan de APSE helpem via configuratiemanagement gereedschap om de verschillende versies van de software te administreren en te beheren. Ook kan de APSE aangeven welke eenheden uit de programmabibliotheek opnieuw moeten worden gecompileerd als in een bepaalde eenheid een wijziging wordt aangebracht. Op het eerste gezicht mag dit niet eenvoudig lijken, maar juist in Ada is dit mogelijk omdat Ada nu eenmaal als regel stelt dat afhankelijkheden tussen programma-eenheden expliciet moeten worden genoemd in de eenheid-contextspecificaties (zie hoofdstuk 20).

23.6 De Fase Van Ingebruikname En Onderhoud



Als de integrale systeemtest succesvol is afgesloten wordt het systeem in gebruik genomen. Tenminste zo zou het horen te zijn, maar meestal begint men al veel eerder met delen van het systeem te gebruiken. Als we echter de ontwerp/codeer/test iteratiemethode hebben toegepast hoeft dit geen bezwaar te zijn, omdat de tot dan toe geteste functies redelijk betrouwbaar zijn. Zoals we al in hoofdstuk 2 aangaven is de onderhoudsfase meestal het kostbaarste gedeelte van de totale levenscyclus. Deze fase is nooit definitief beëindigd, hoogstens nemen de werkzaamheden geleidelijk af. In bepaalde gevallen kan op den duur blijken dat de apparatuur zodanig is verouderd en dat de programmatuur zo weinig overdraagbaar is, dat men beslist het totale systeem 'op de schroothoop te werpen', maar meestal wordt vanuit een kleiner systeem een groter systeem ontwikkeld.

Ook tijdens het onderhoud moet het ontwerp/codeer/test proces worden uitgevoerd en dit kan op dezelfde manier met gebruik van Ada als eerder beschreven. Omdat de ontwerptaal hier niet verschilt van de implementatietaal is bij het aanbrengen van een verandering de documentatie automatisch aangepast. Wegens de gestructureerdheid van Ada is het bij het aanbrengen van wijzigingen niet moeilijk de oorspronkelijke structuur intact te laten. Ook hier kan APSE weer helpen bij het configuratiemanagement.



Ada op haar sterfbed; een schets door haar moeder.
(Bodleian Library, Oxford.)

We hebben hier maar een beknopt overzicht gegeven van de totale software levenscyclus; voor een complete beschrijving zouden meerdere boekdelen nodig zijn. Wel hebben we laten zien dat Ada tijdens de gehele levenscyclus kan worden gebruikt en een evenwichtig ontwikkelingsproces kan ondersteunen.

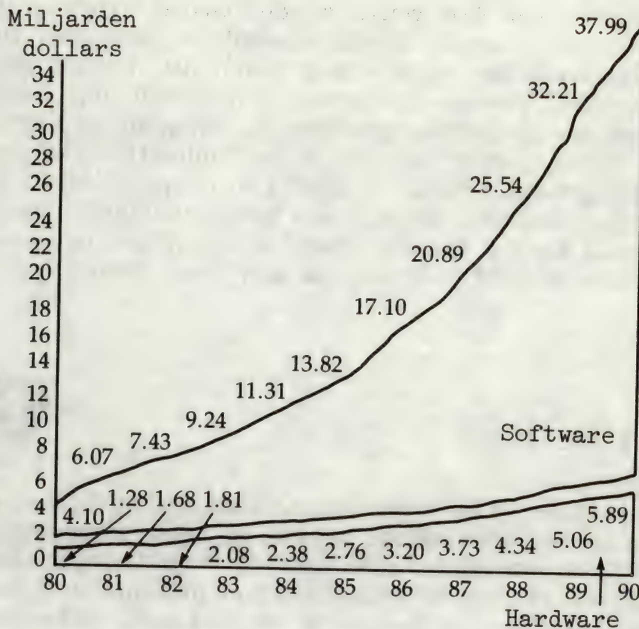
24 TOEKOMSTIGE ONTWIKKELINGEN EN CONCLUSIES

Zo zijn we aan het einde gekomen van onze behandeling van de programmeertaal Ada. We kunnen op het moment dat we dit schrijven niet voorspellen of Ada uiteindelijk succes zal hebben als een gereedschap voor het ontwikkelen van betrouwbare en onderhoudbare software. Het succes van Ada zal voor een belangrijk deel afhangen van de wijze waarop de taal gebruikt gaat worden; de slotconlcusie zullen we aan de historici overlaten. In dit laatste hoofdstuk zullen we echter nog wel aantonen dat er talrijke aanwijzigingen zijn, dat Ada van grote invloed zal zijn op de ontwikkeling van de computerwetenschap in de komende jaren.

24.1 Ontwikkelingen In DoD Software  

In hoofdstuk 2 bespraken we de hardware/software kostenverhouding, zoals deze bleek uit een rapport uit het begin van de jaren 70. Figuur 24-1 geeft een gedetailleerder overzicht uit een in 1980 gepubliceerd onderzoek [1]. De grafiek geeft aan, dat verwacht wordt dat de softwarekosten voor DoD dramatisch zullen stijgen. Deze toename in kosten is voornamelijk te wijten aan een verwachte grote toename in het aantal toepassingen. Vanaf 1985 zal DoD overgaan op Ada voor alle nieuwe ingebedde software computertoepassingen. Op den duur zullen de bestedingen van DoD aan Ada software de totale uitgaven voor FORTRAN en COBOL software, tot nu toe door DoD gemaakt, verre overtreffen.

De toenemende softwarekosten gaan samen met een verbetering in de verhouding tussen prijs en prestatie van de hardware. Omstreeks het jaar 2000 zullen er dus naar verwachting veel meer toepassingen voor embedded software mogelijk zijn, omdat dergelijke oplossingen steeds meer kosteneffectief zullen worden. We hebben gezien dat Ada een uitstekende taal is voor het ontwikkelen van dergelijke systemen en we verwachten dan ook dat steeds ingewikkelder problemen met behulp van Ada zullen kunnen worden aangepakt.



Figuur 24-1 DoD hardware/software kostenverhouding.

24.2 Het Succes Van Ada





Sinds de eerste FORTRAN versie zijn er letterlijk honderden nieuwe talen voorgesteld en gerealiseerd en er zullen we nog honderden komen - één daarvan zal ook de plaats van Ada innemen. Toch zijn om tal van redenen maar weinig van de nieuwe programmeertalen in brede kring geaccepteerd. Wegner [2] geeft de volgende vergelijking als criterium voor het succes van een taal:

SUCCES := ONTWERP + IMPLEMENTATIE + ONDERSTEUNING +
+ BEHOEFTE

Als deze vergelijking wordt toegepast op Ada dan valt in te zien, dat Ada inderdaad in al deze opzichten een 'succesvolle' taal is. In hoofdstuk 3 lieten we zien dat Ada een taal is met een goede ontwerp-filosofie, die zich richt op het gebruik van moderne software ontwerp-methoden. Door middel van de STEELMAN specificaties, gevolgd door een test- en een evaluatiefase, is het aantal ontwerpfouten gering. Compilerbouwtechnieken hebben zich enorm ontwikkeld sinds de begintijd van FORTRAN en COBOL; de problemen bij het ontwikkelen van efficiënte vertalers zijn nu goed gedefinieerd en grotendeels opgelost.

De introductie van Ada wordt zonder twijfel krachtig ondersteund door het Amerikaanse Departement van Defensie. Ook COBOL kreeg die ondersteuning en deze taal wordt nu in ieder geval zeer algemeen gebruikt. Verdere ondersteuning wordt nog geboden door de industrie en academische instellingen, zowel in de Verenigde Staten als in Europa. Ada voorziet in een behoefte, zoals geen enkele andere programmeertaal: de STEELMAN specificaties hebben er toe geleid dat de eisen die aan een gemeenschappelijke hogere programmeertaal moeten worden gesteld, exact werden omschreven en in bijna ieder opzicht voldoet Ada aan deze eisen.

24.3 Tot Besluit





Het zou te ver gaan te beweren dat Ada de softwarecrisis oplost. Ook is Ada niet 'het allerlaatste woord' op programmeertaalgebied - een dergelijk onbereikbaar doel hebben de ontwerpers ook nooit voor ogen gehad. Het voordeel dat uit het gebruik van Ada te trekken valt ligt eigenlijk niet zo zeer in de taal zelf, maar veel meer in het feit dat Ada goede software ontwikkelingstechnieken kan bevorderen. Net als bij ander gereedschap geldt voor Ada, dat de taal ofwel correct gebruikt kan worden om complexe problemen eenvoudig op te lossen, ofwel incorrect door probleemoplossingen nodeloos complex te maken.

Volgens hedendaagse maatstaven is Ada de beste keuze als het gaat om een taal voor het ontwerpen en het bouwen van grote realtime systemen, maar ook in andere probleemgebieden kan Ada met succes worden toegepast. Toch stelt Whitaker: "Nieuwe omgevingen, nieuwe apparatuur en nieuwe ontwikkelingen in de informatica zullen uiteindelijk ook de beste programmeertaal doen verouderen" [3]. Desalniettemin verwachten wij dat Ada's levenscyclus zich tot een niet gering aantal jaren van de volgende eeuw zal uitstrekken en dat de taal door het DoD voor de meeste toepassingen gebruikt gaat worden.

Bij wijze van waarschuwing voor Ada-gebruikers vermelden we een opmerking van Fisher, die stelt dat "de belangrijkste resultaten zijn te verwachten van betere programmeermethoden en -technieken, meer gemeenschappelijk te gebruiken software en bruikbaarder en gemakkelijker toegankelijke software ontwikkelingsgereedschappen" [4]. Ada op zichzelf is geen voldoende garantie voor de ontwikkeling van betere programma's, maar gecombineerd met een gedisciplineerde ontwerpmethode kan de taal ons helpen bij het beheersen van complexe oplossingen en leiden tot begrijpelijke, betrouwbare, onderhoudbare en efficiënte systemen.

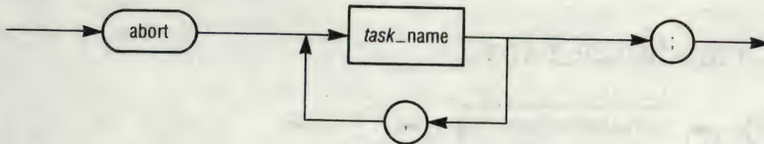
APPENDIX A: ADA SYNTAXDIAGRAMMEN

Van de verschillende methodes om een syntax van een programmeertaal formeel te definiëren is Bachus-Naur Form (BNF) de meest gebruikte vanwege de beknopte notatiewijze. De BNF produktieregels zijn echter voor niet ingewijden lastig leesbaar en daarom maken wij hier voor de beschrijving van Ada gebruik van syntaxdiagrammen; in feite grafische weergaven van de BNF notatie.

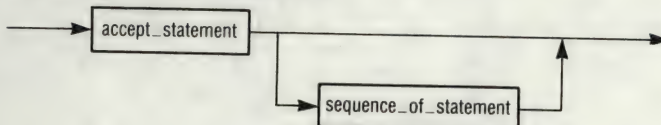
Syntaxdiagrammen worden van links naar rechts gelezen, waarbij de richting van de pijlen wordt gevolgd. Een pijl kan ook terug naar links verwijzen en een lus vormen: daarmee wordt aangegeven dat de constructie herhaald kan worden. Een woord in een rechthoekje is een constructie die in een ander diagram wordt gedefinieerd (een niet terminale produktie). Een woord in een cirkel of in een ellipsachtige figuur geeft een string van tekens weer die letterlijk zo in de taal voorkomt. Een constructie kan worden voorafgegaan door een cursief geschreven woord; een dergelijke constructie is equivalent met een constructie zonder dit cursieve woord. Dit is slechts toegevoegd om de betekenis te verduidelijken.

We geven de syntaxdiagrammen in alfabetische volgorde. We hebben alle constructies opgenomen, met als uitzondering <basic_character>, <digit>, <basic_graphic_character> en <graphic_character>, waarvan de betekenis voor de hand ligt. De diagrammen zijn verder zo getekend, dat de gewenste lay-out binnen een programma van de gedefinieerde constructie er uit blijkt.

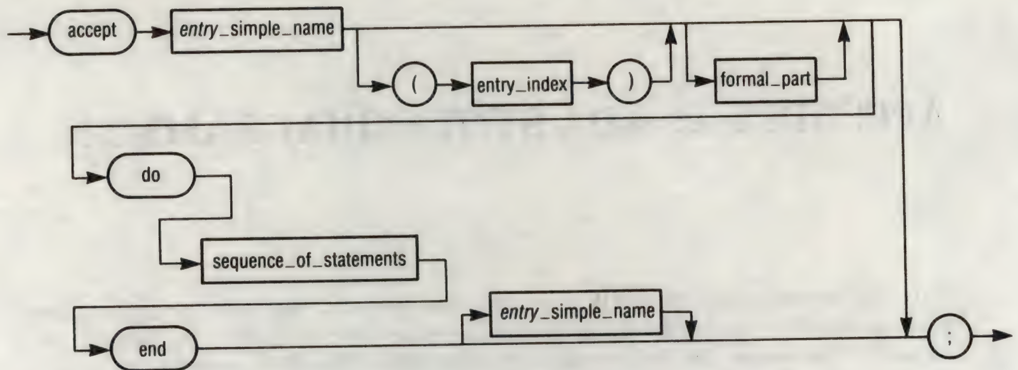
ABORT_STATEMENT



ACCEPT_ALTERNATIVE



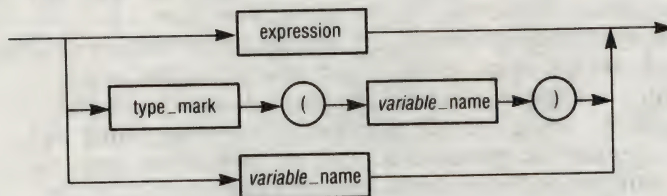
ACCEPT_STATEMENT



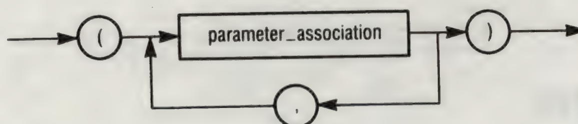
ACCESS_TYPE_DEFINITION



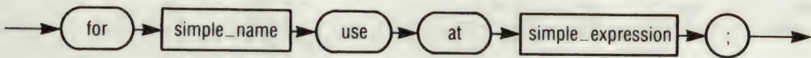
ACTUAL_PARAMETER



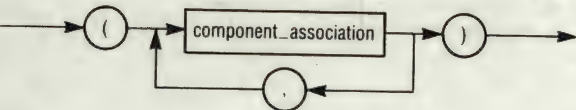
ACTUAL_PARAMETER_PART



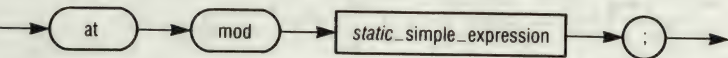
ADDRESS_CLAUSE



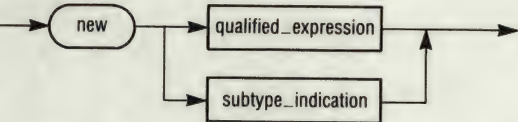
AGGREGATE



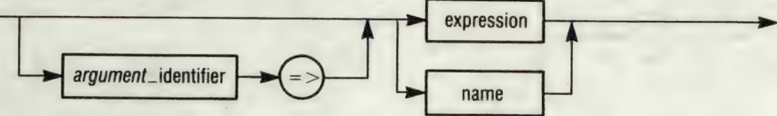
ALIGNMENT_CLAUSE



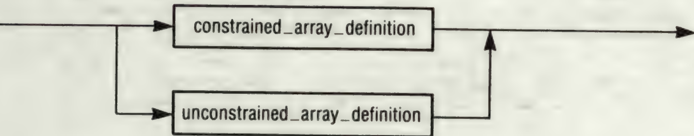
ALLOCATOR



ARGUMENT_ASSOCIATION



ARRAY_TYPE_DEFINITION



ASSIGNMENT_STATEMENT

*variable*_name → := → expression → ;

ATTRIBUTE

prefix → ' → attribute_designator

ATTRIBUTE_DESIGNATOR

simple_name → (→ *universal_static*_expression →)

BASE

integer

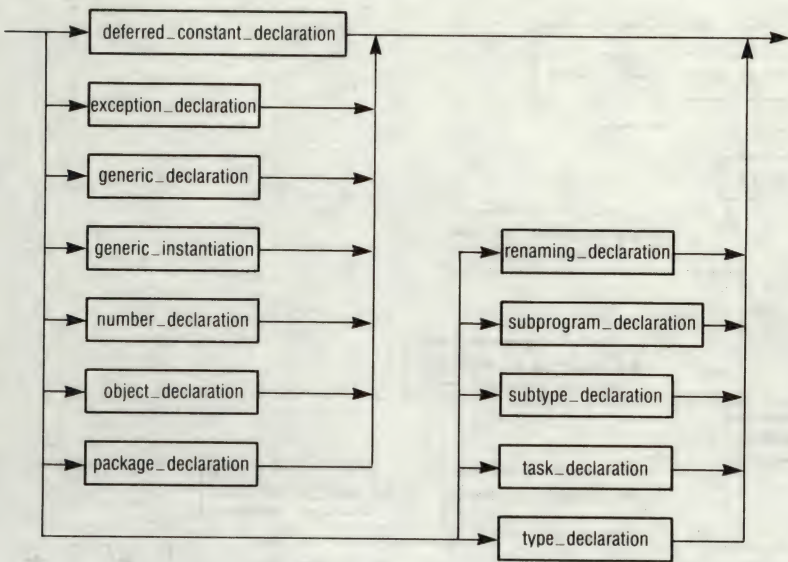
BASED_INTEGER

extended_digit ← _

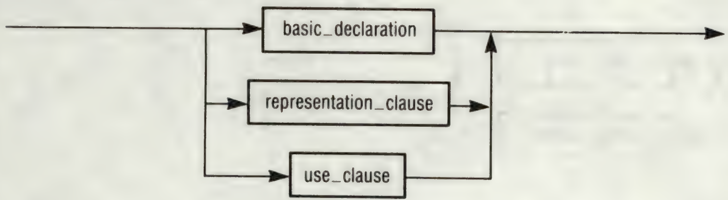
BASED_LITERAL

base → # → based_integer → . → based_integer → # → exponent

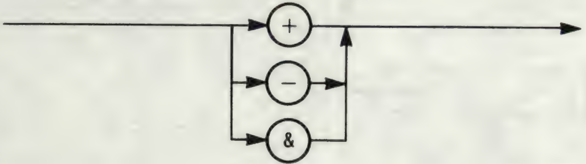
BASIC_DECLARATION



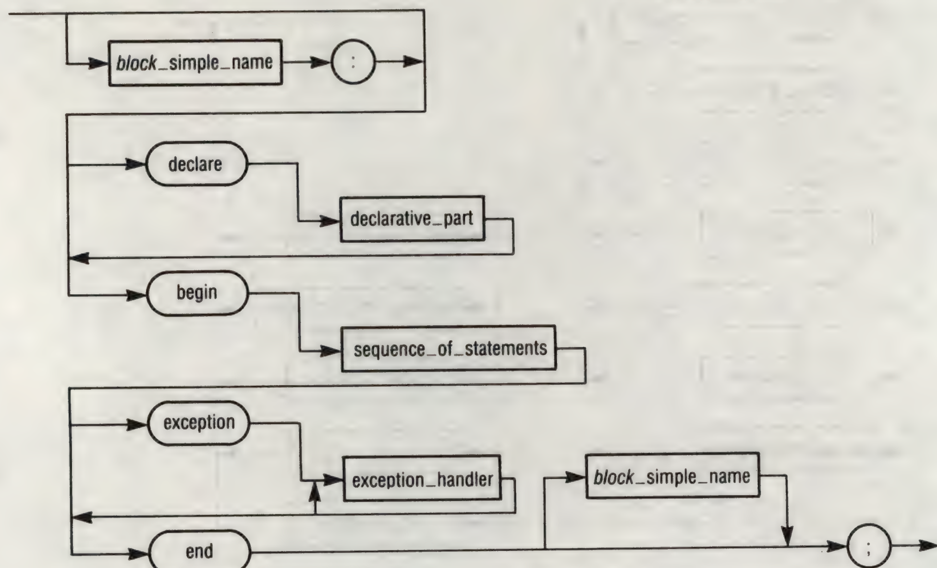
BASIC_DECLARATIVE_ITEM



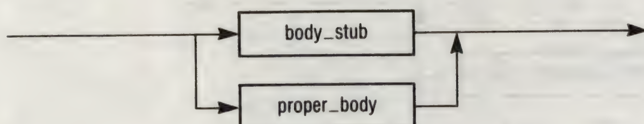
BINARY_ADDING_OPERATOR



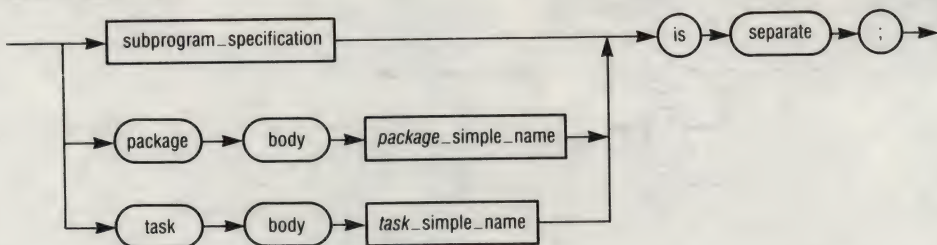
BLOCK_STATEMENT



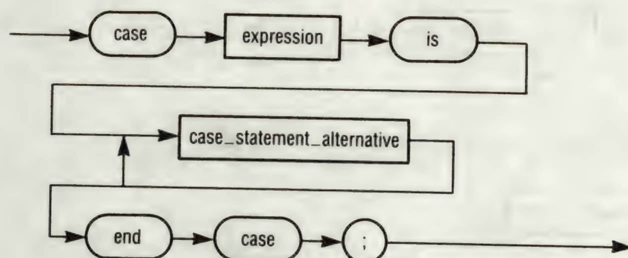
BODY



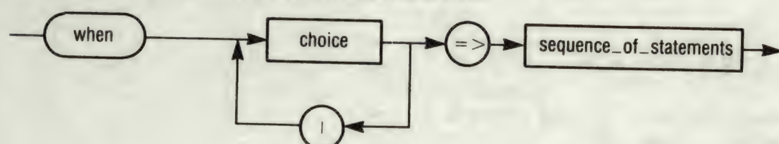
BODY_STUB



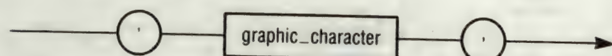
CASE_STATEMENT



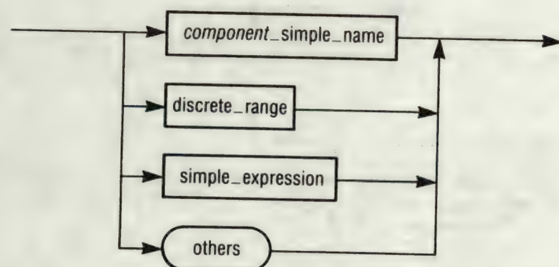
CASE_STATEMENT_ALTERNATIVE



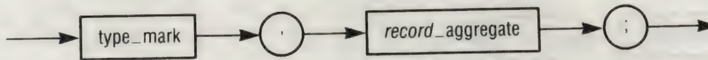
CHARACTER_LITERAL



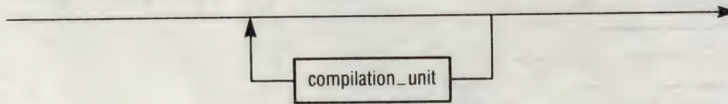
CHOICE



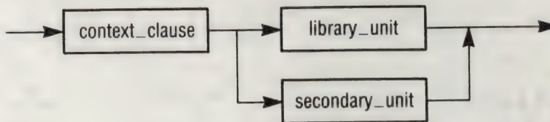
CODE_STATEMENT



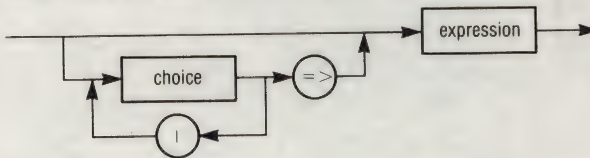
COMPILATION



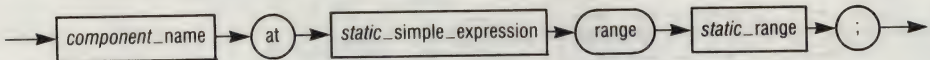
COMPILATION_UNIT



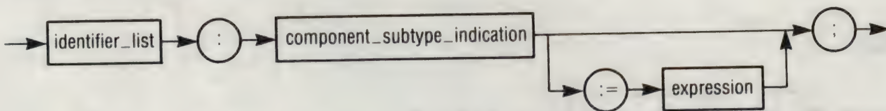
COMPONENT_ASSOCIATION



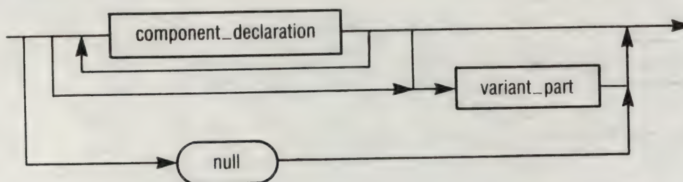
COMPONENT_CLAUSE



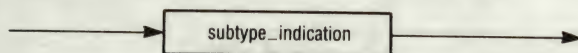
COMPONENT_DECLARATION



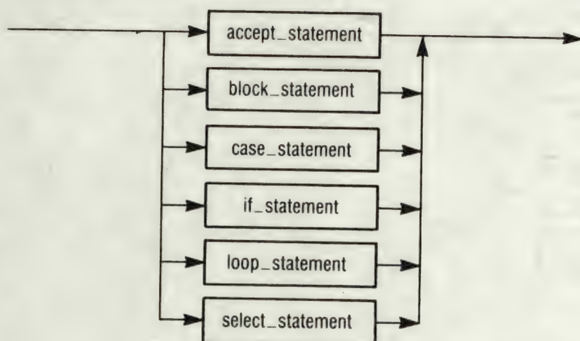
COMPONENT_LIST



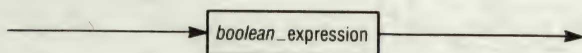
COMPONENT_SUBTYPE_DEFINITION



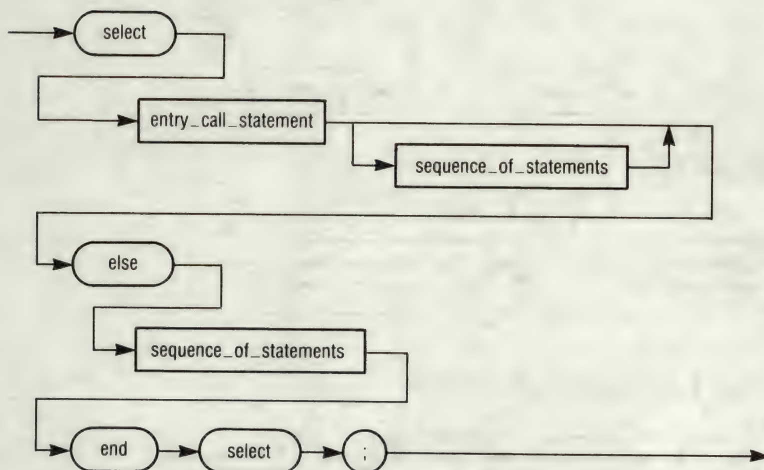
COMPOUND_STATEMENT



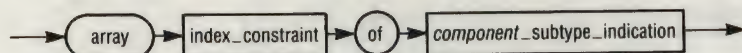
CONDITION



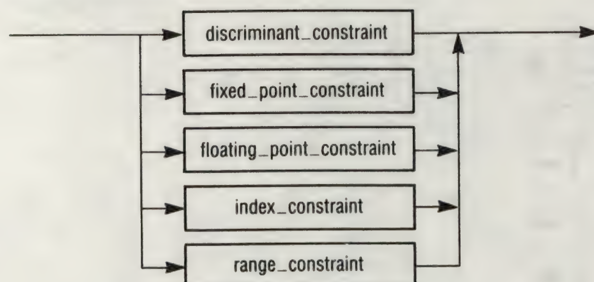
CONDITIONAL_ENTRY_CALL



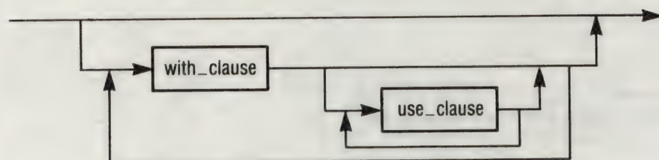
CONSTRAINED_ARRAY_DEFINITION



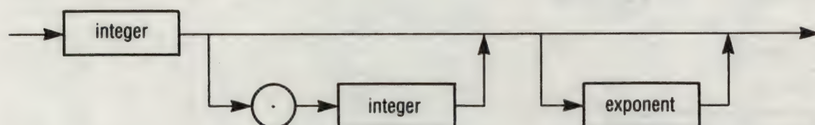
CONSTRAINT



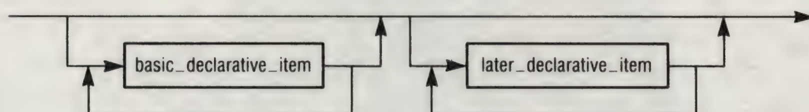
CONTEXT_CLAUSE



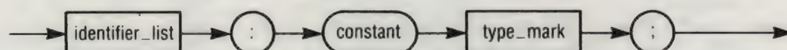
DECIMAL_LITERAL



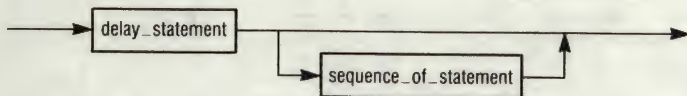
DECLARATIVE_PART



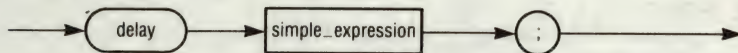
DEFERRED_CONSTANT_DECLARATION



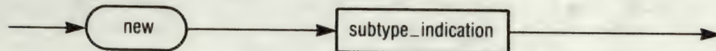
DELAY_ALTERNATIVE



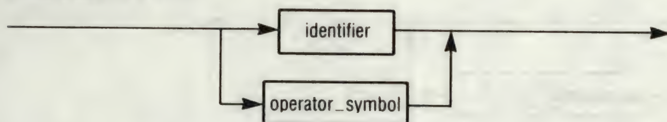
DELAY_STATEMENT



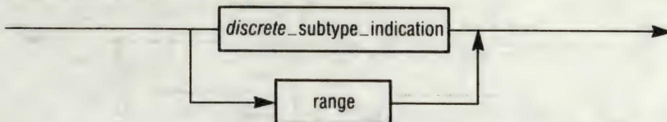
DERIVED_TYPE_DEFINITION



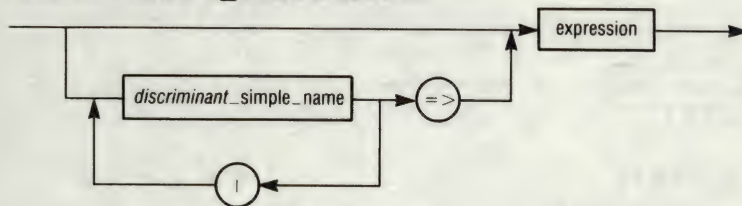
DESIGNATOR



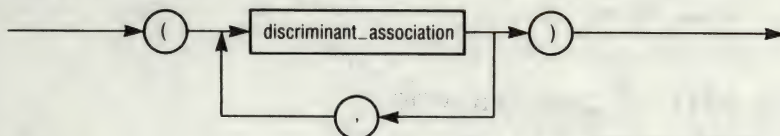
DISCRETE_RANGE



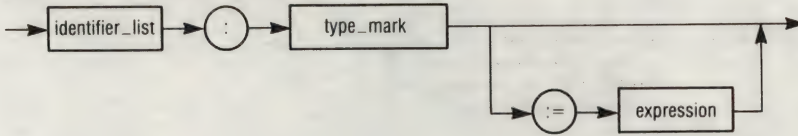
DISCRIMINANT_ASSOCIATION



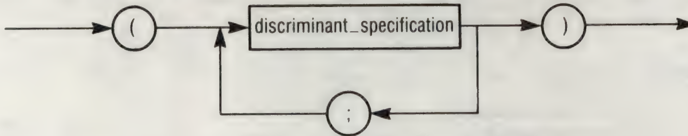
DISCRIMINANT_CONSTRAINT



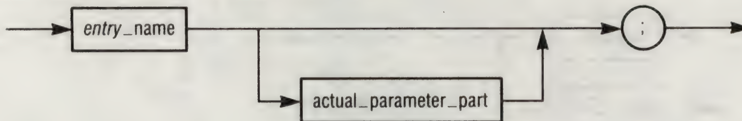
DISCRIMINANT_SPECIFICATION



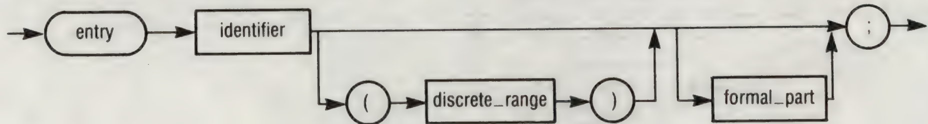
DISCRIMINANT_PART



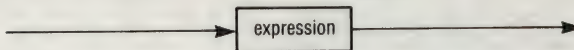
ENTRY_CALL_STATEMENT



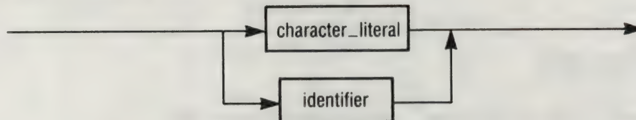
ENTRY_DECLARATION



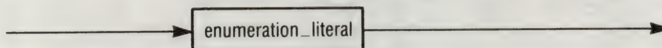
ENTRY_INDEX



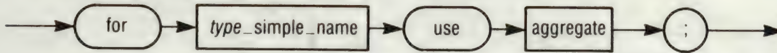
ENUMERATION_LITERAL



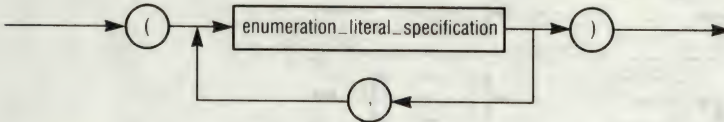
ENUMERATION_LITERAL_SPECIFICATION



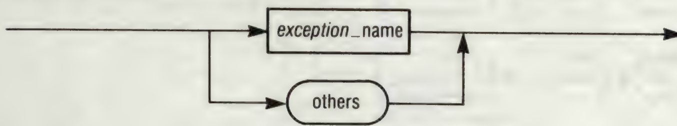
ENUMERATION_REPRESENTATION_CLAUSE



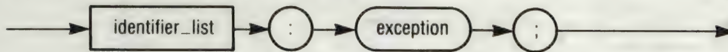
ENUMERATION_TYPE_DEFINITION



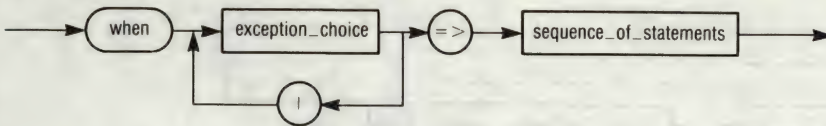
EXCEPTION_CHOICE



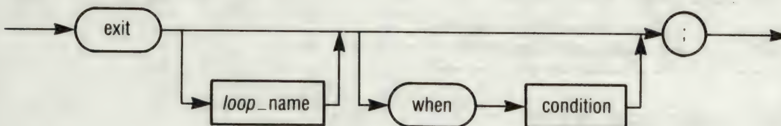
EXCEPTION_DECLARATION



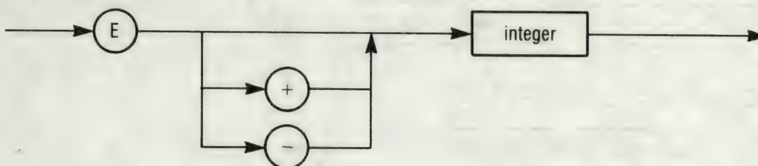
EXCEPTION_HANDLER



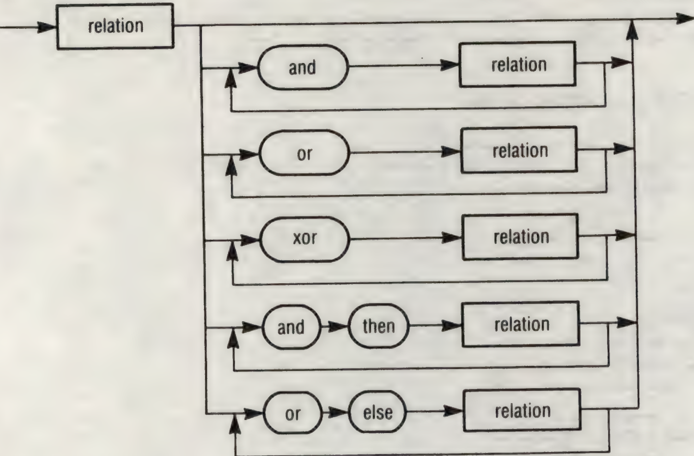
EXIT_STATEMENT



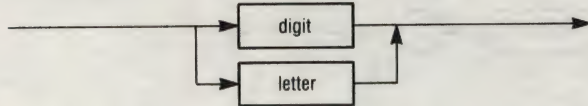
EXPONENT



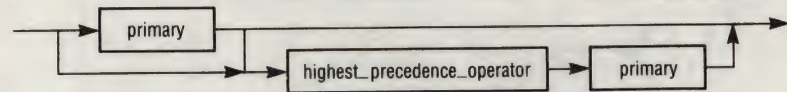
EXPRESSION



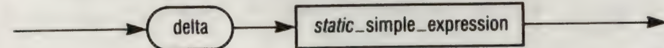
EXTENDED_DIGIT



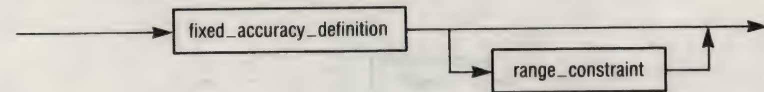
FACTOR



FIXED_ACCURACY_DEFINITION



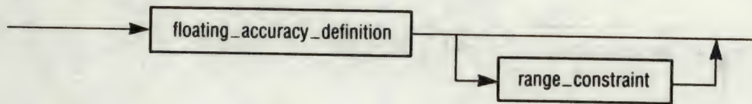
FIXED_POINT_CONSTRAINT



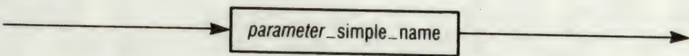
FLOATING_ACCURACY_DEFINITION



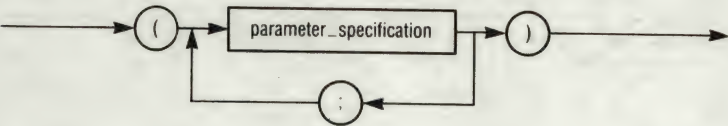
FLOATING_POINT_CONSTRAINT



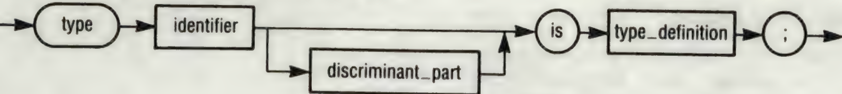
FORMAL_PARAMETER



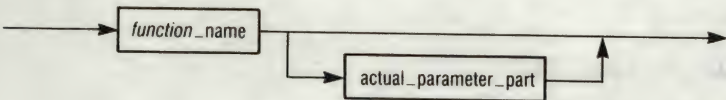
FORMAL_PART



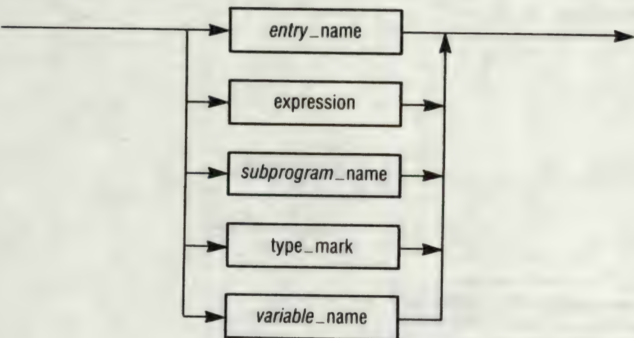
FULL_TYPE_DECLARATION



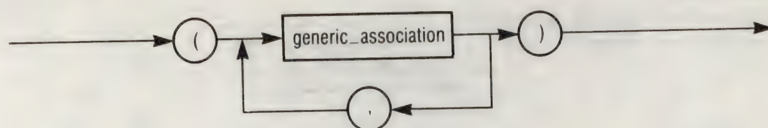
FUNCTION_CALL



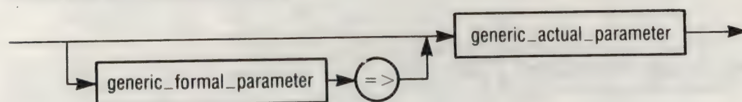
GENERIC_ACTUAL_PARAMETER



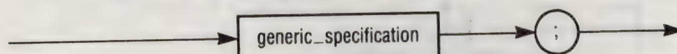
GENERIC_ACTUAL_PART



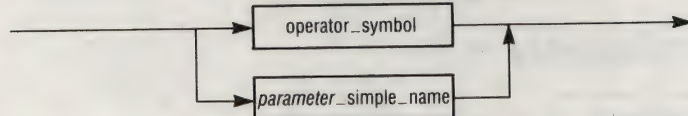
GENERIC_ASSOCIATION



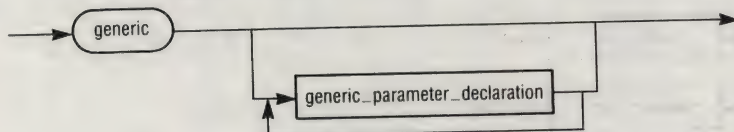
GENERIC_DECLARATION



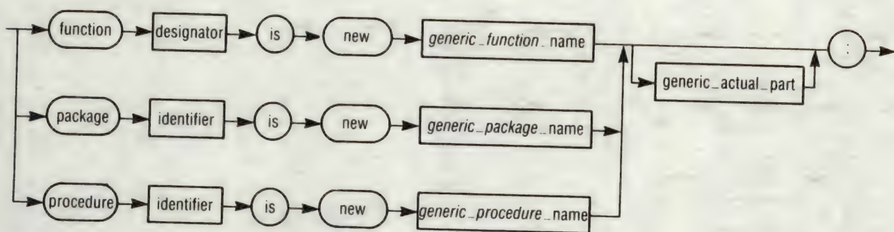
GENERIC_FORMAL_PARAMETER



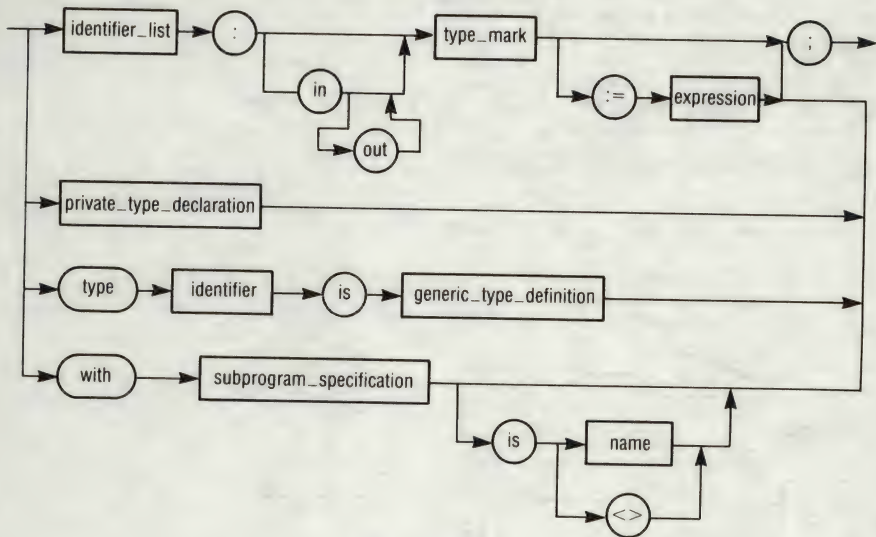
GENERIC_FORMAL_PART



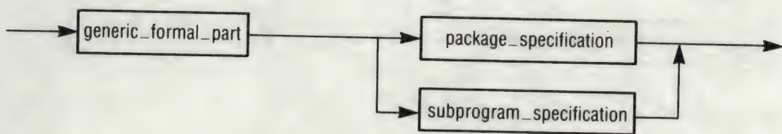
GENERIC_INSTANTIATION



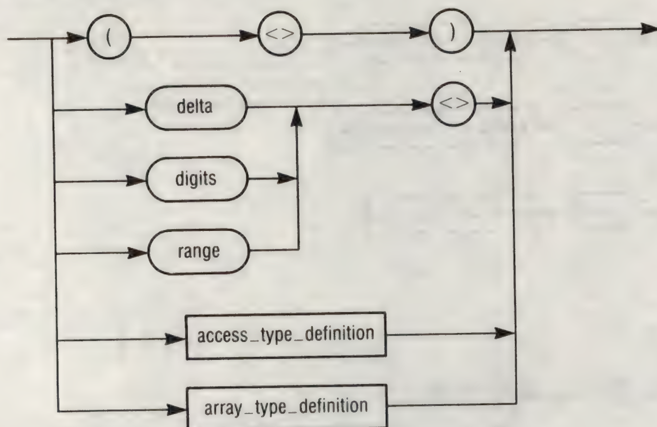
GENERIC_PARAMETER_DECLARATION



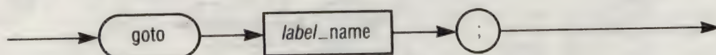
GENERIC_SPECIFICATION



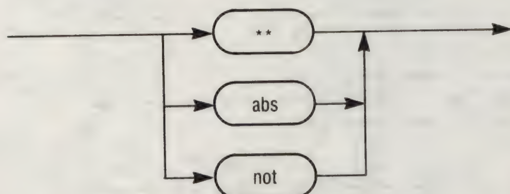
GENERIC_TYPE_DEFINITION



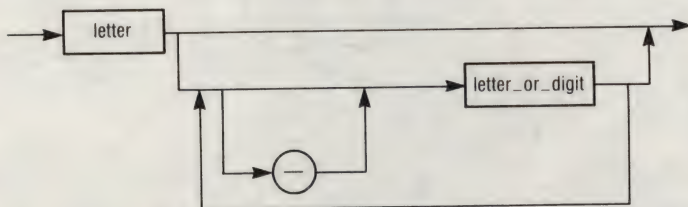
GOTO_STATEMENT



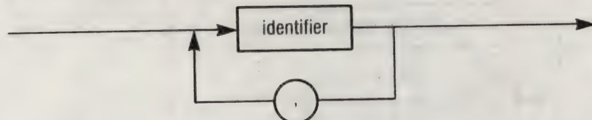
HIGHEST_PRECEDENCE_OPERATOR



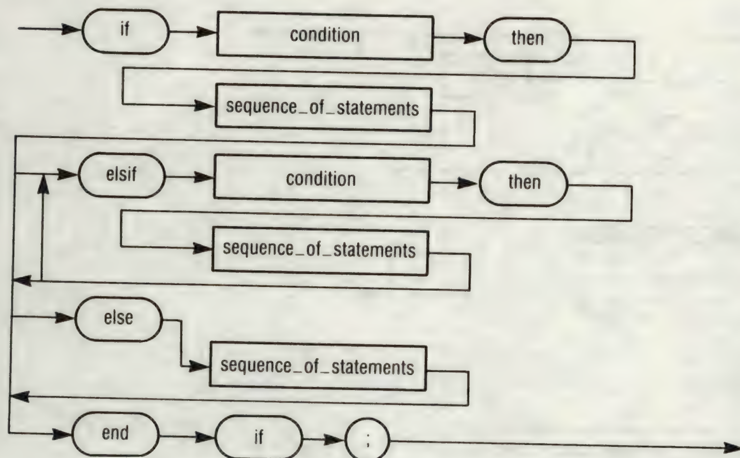
IDENTIFIER



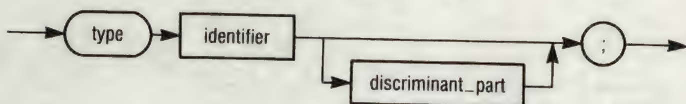
IDENTIFIER_LIST



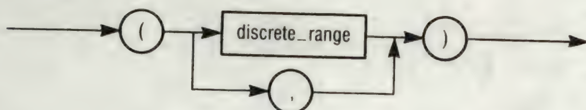
IF_STATEMENT



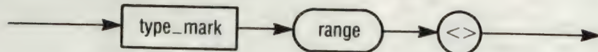
INCOMPLETE_TYPE_DECLARATION



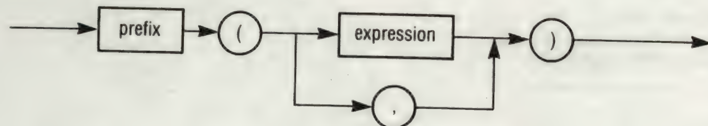
INDEX_CONSTRAINT



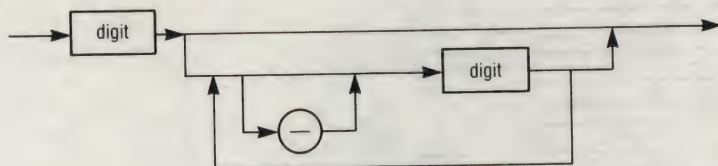
INDEX_SUBTYPE_DEFINITION



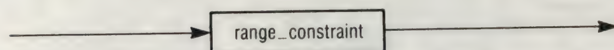
INDEXED_COMPONENT



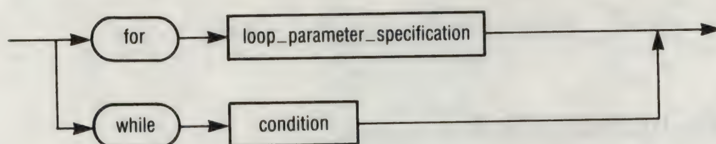
INTEGER



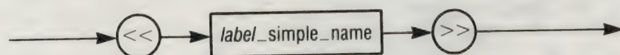
INTEGER_TYPE_DEFINITION



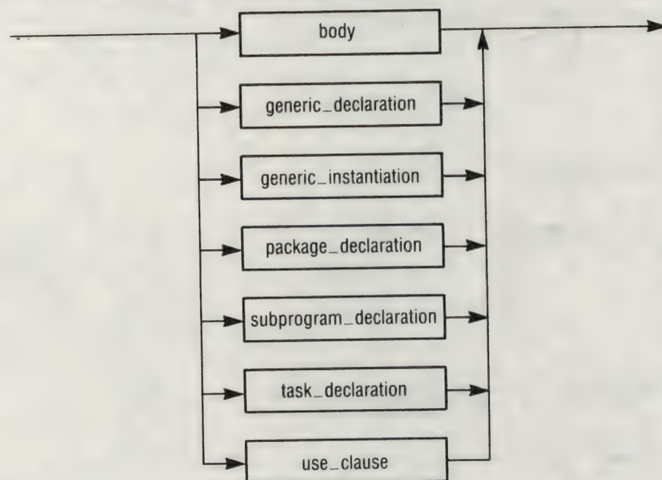
ITERATION_SCHEME



LABEL



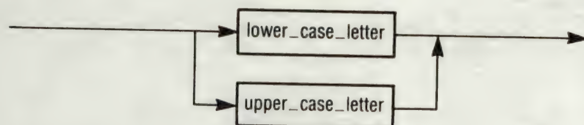
LATER_DECLARATIVE_ITEM



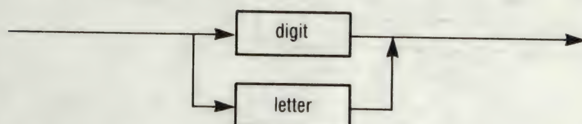
LENGTH_CLAUSE



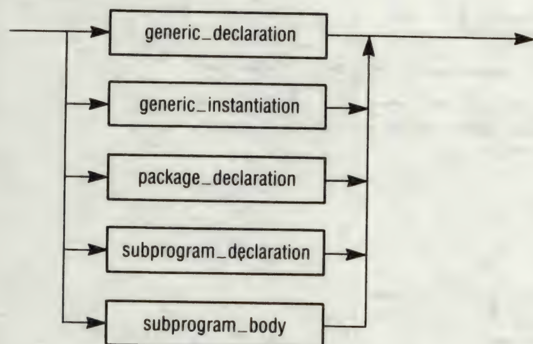
LETTER



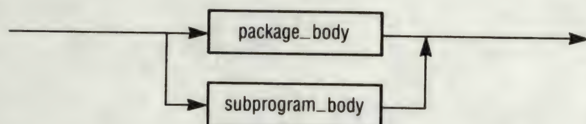
LETTER_OR_DIGIT



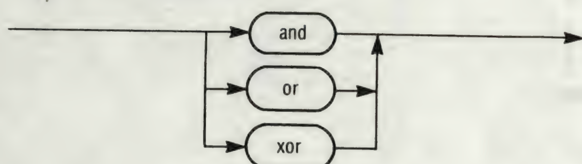
LIBRARY_UNIT



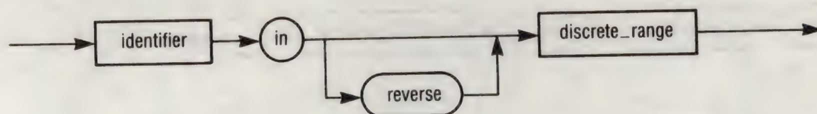
LIBRARY_UNIT_BODY



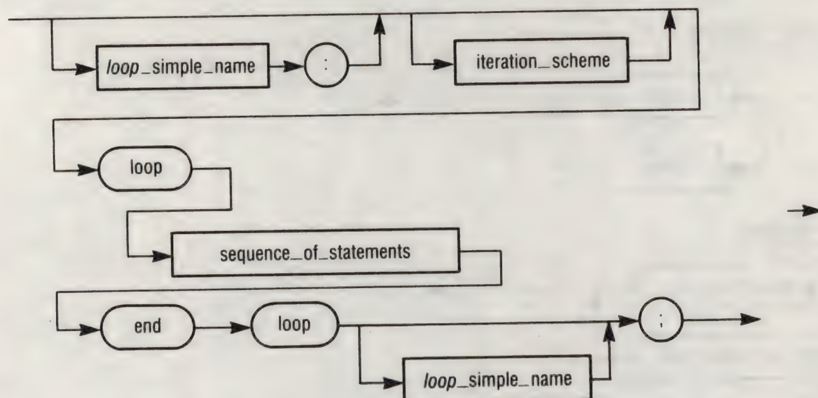
LOGICAL_OPERATOR



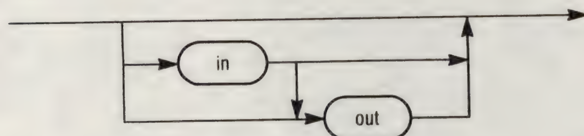
LOOP_PARAMETER_SPECIFICATION



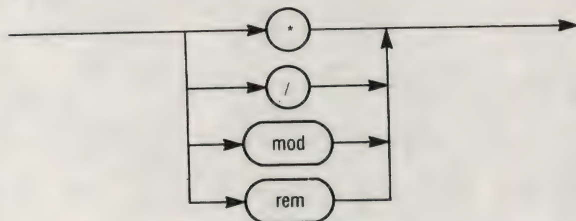
LOOP_STATEMENT



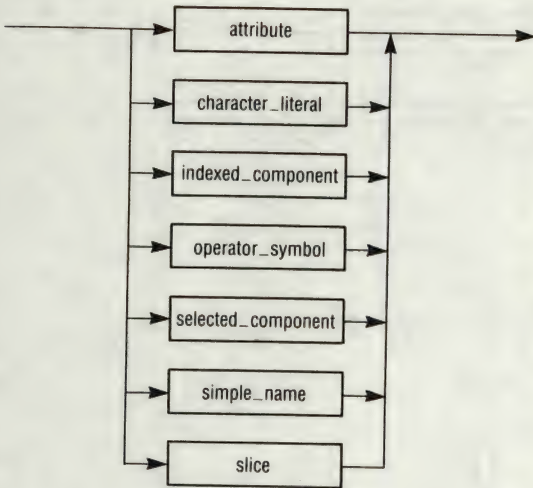
MODE



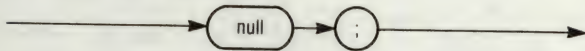
MULTIPLYING_OPERATOR



NAME



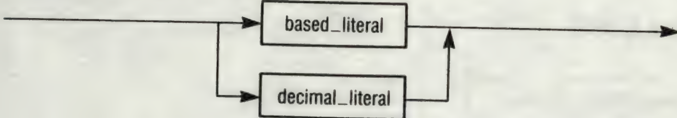
NULL_STATEMENT



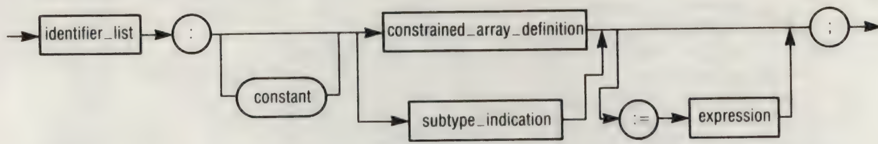
NUMBER_DECLARATION



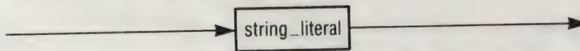
NUMERIC_LITERAL



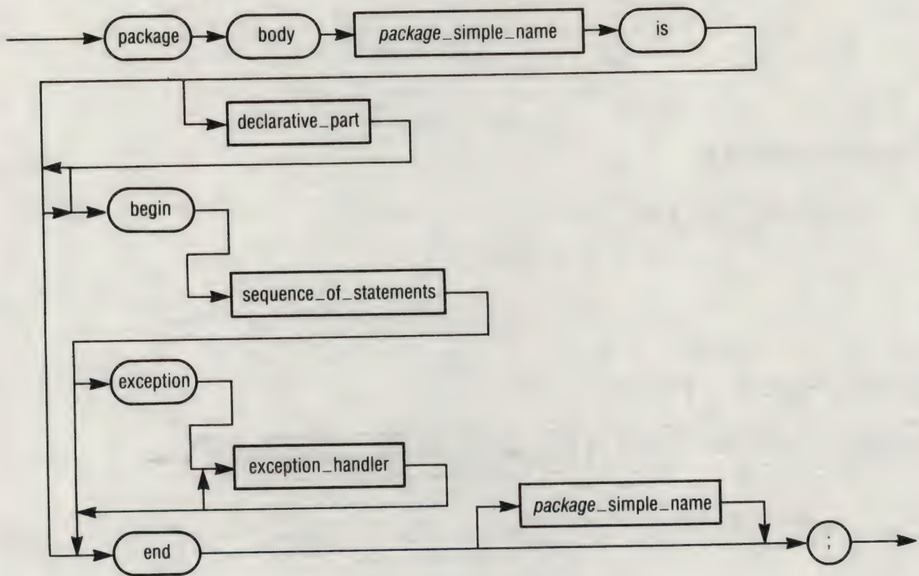
OBJECT_DECLARATION



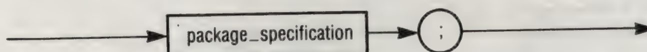
OPERATOR_SYMBOL



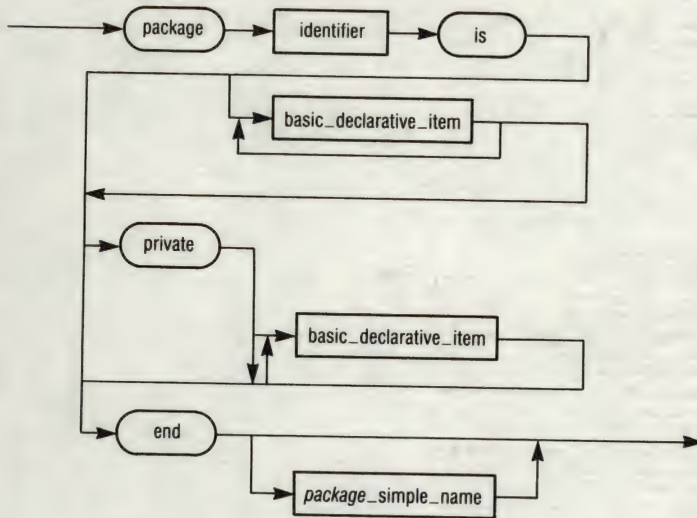
PACKAGE_BODY



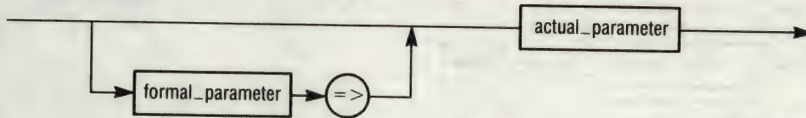
PACKAGE_DECLARATION



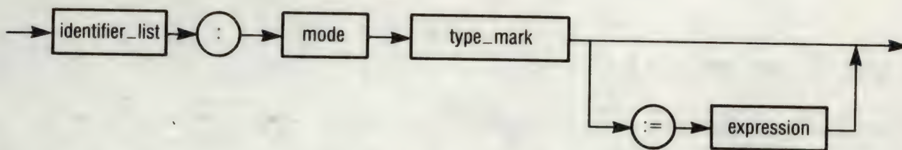
PACKAGE_SPECIFICATION



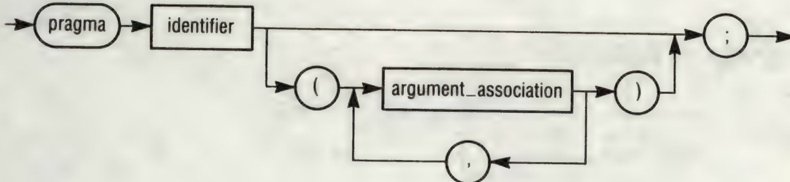
PARAMETER_ASSOCIATION



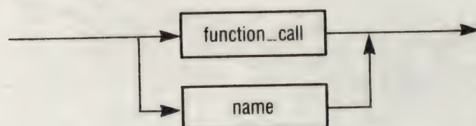
PARAMETER_SPECIFICATION



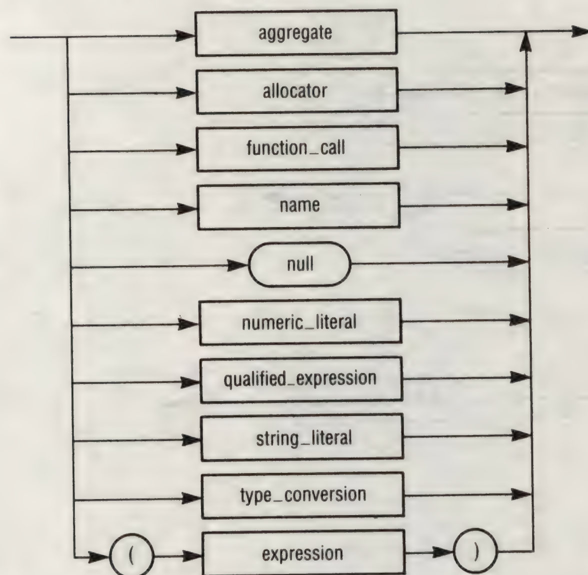
PRAGMA



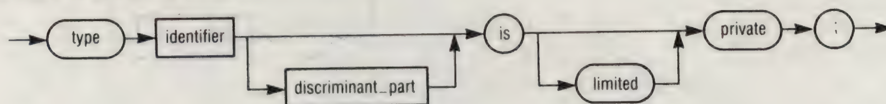
PREFIX



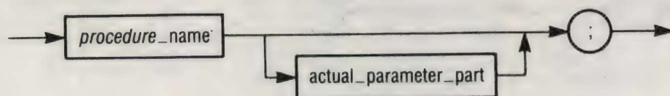
PRIMARY



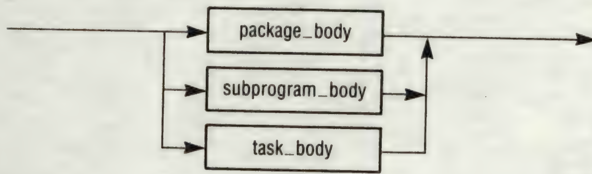
PRIVATE_TYPE_DECLARATION



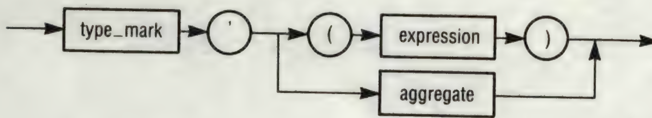
PROCEDURE_CALL_STATEMENT



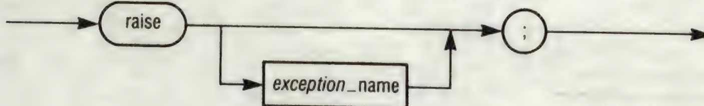
PROPER_BODY



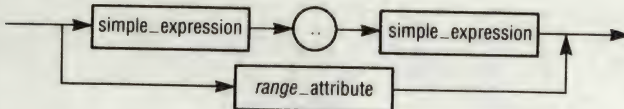
QUALIFIED_EXPRESSION



RAISE_STATEMENT



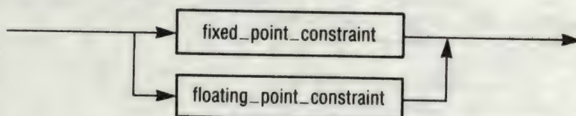
RANGE



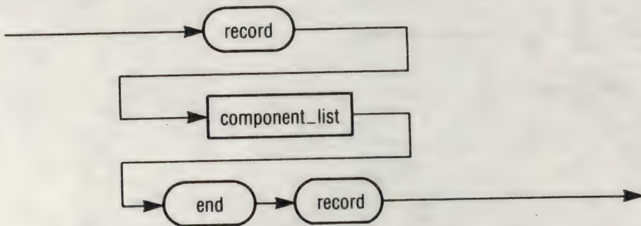
RANGE_CONSTRAINT



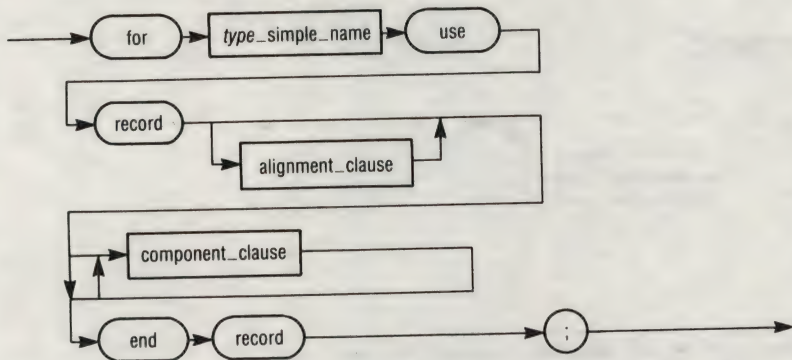
REAL_TYPE_DEFINITION



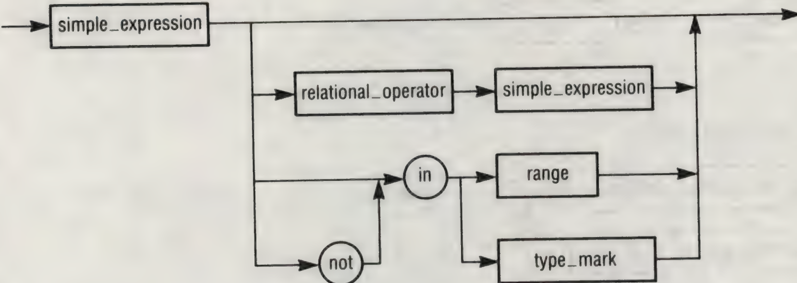
RECORD_TYPE_DEFINITION



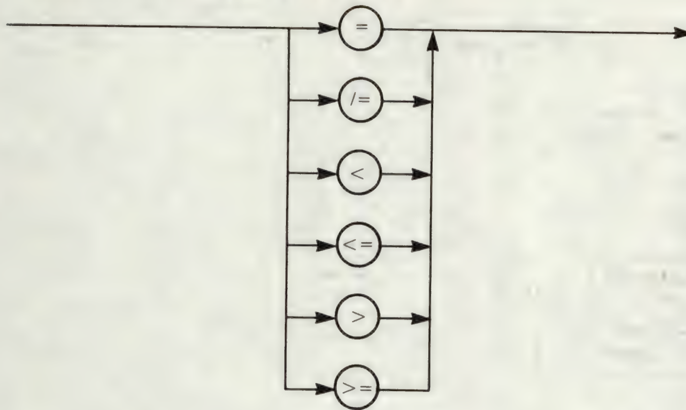
RECORD_REPRESENTATION_CLAUSE



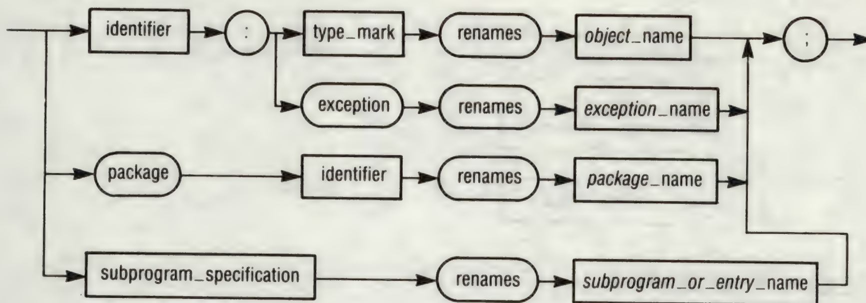
RELATION



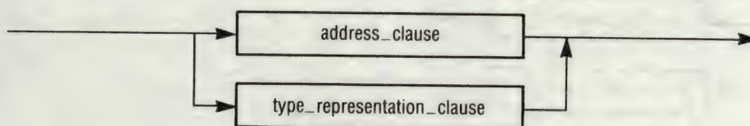
RELATIONAL_OPERATOR



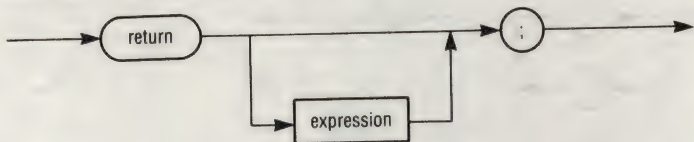
RENAMING_DECLARATION



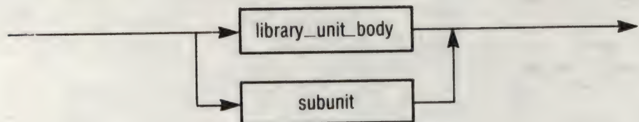
REPRESENTATION_CLAUSE



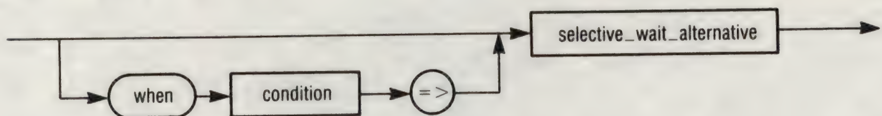
RETURN_STATEMENT



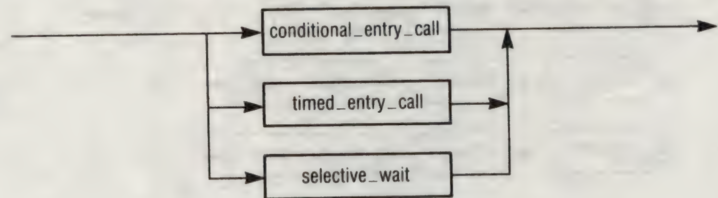
SECONDARY_UNIT



SELECT_ALTERNATIVE



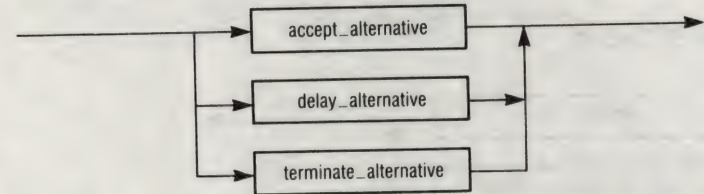
SELECT_STATEMENT



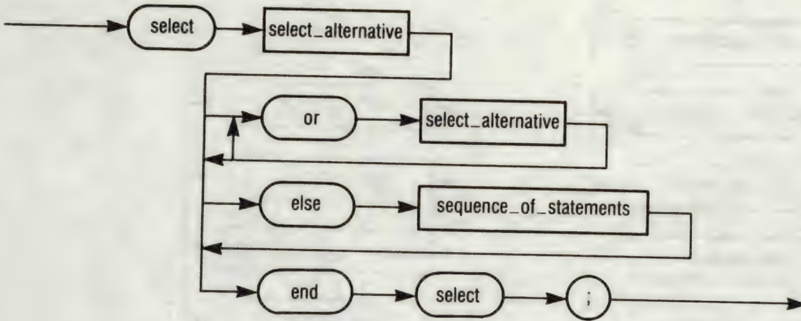
SELECTED_COMPONENT



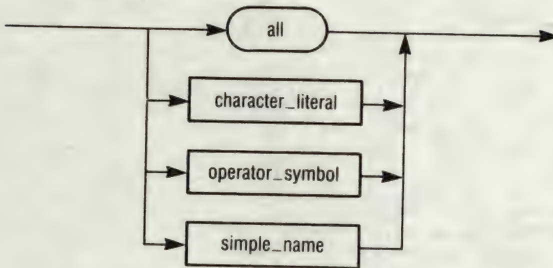
SELECTIVE_WAIT_ALTERNATIVE



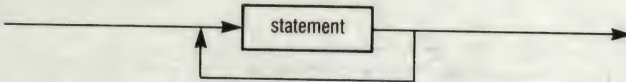
SELECTIVE_WAIT



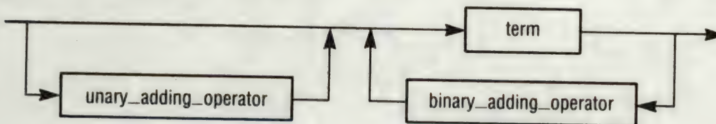
SELECTOR



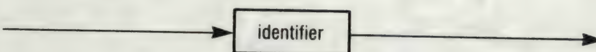
SEQUENCE_OF_STATEMENTS



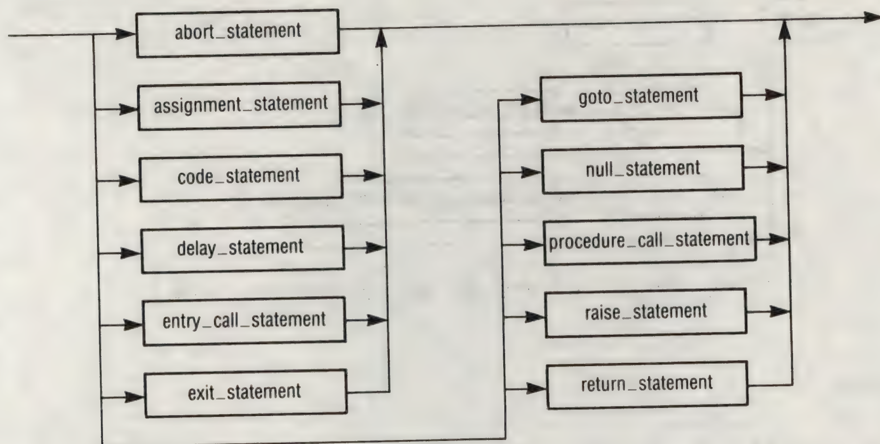
SIMPLE_EXPRESSION



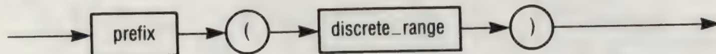
SIMPLE_NAME



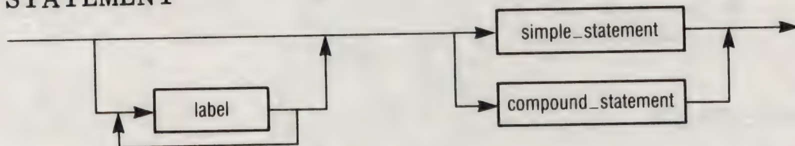
SIMPLE_STATEMENT



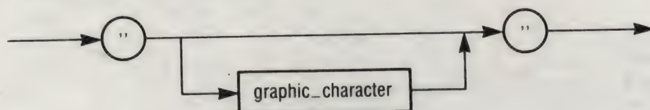
SLICE



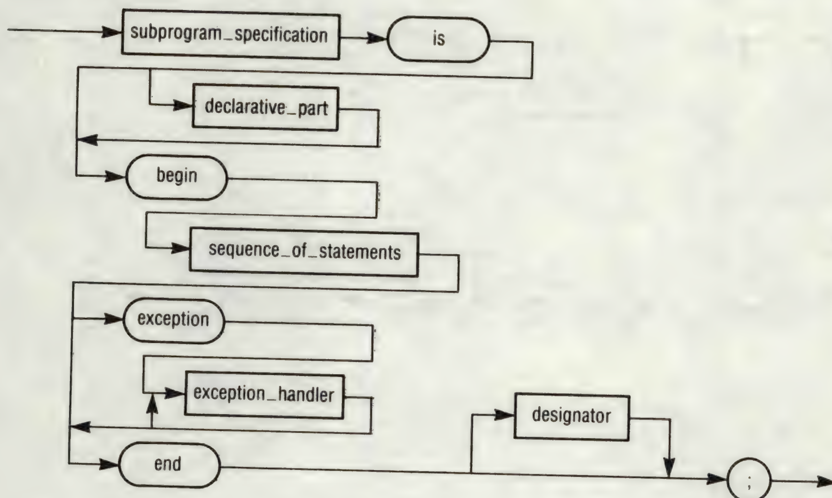
STATEMENT



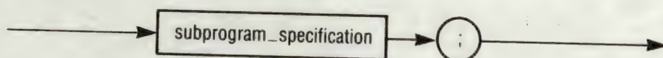
STRING_LITERAL



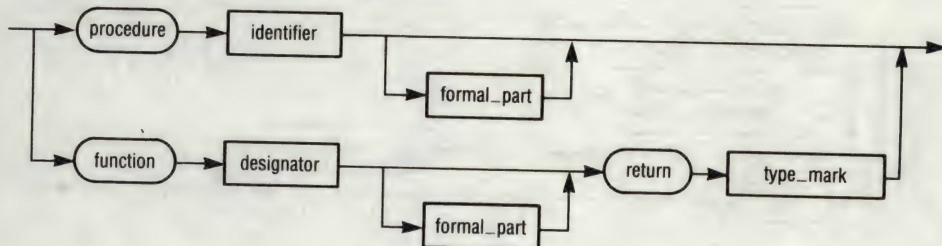
SUBPROGRAM_BODY



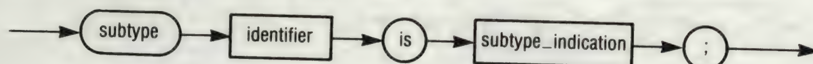
SUBPROGRAM_DECLARATION



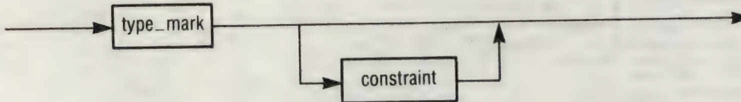
SUBPROGRAM_SPECIFICATION



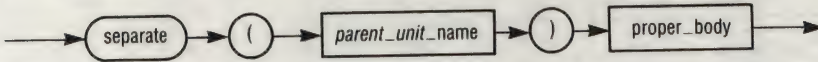
SUBTYPE_DECLARATION



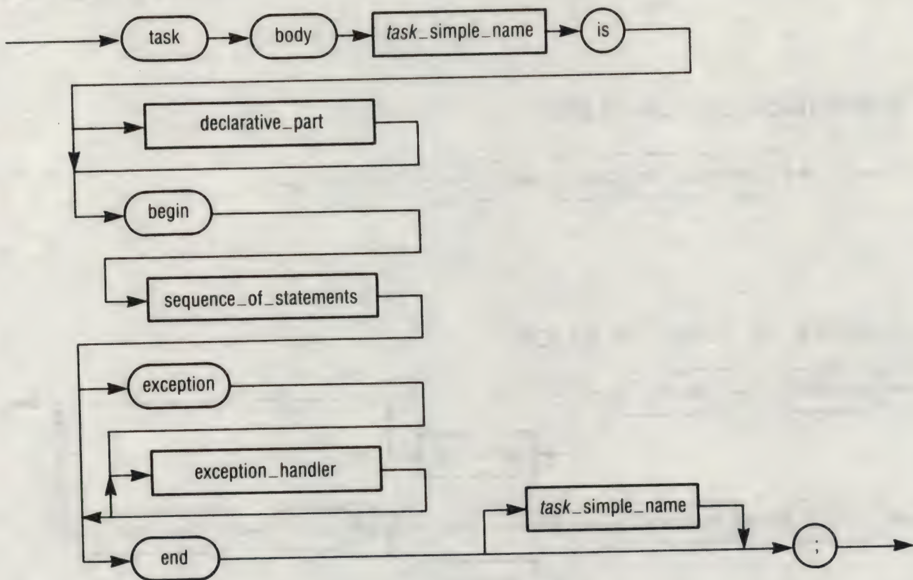
SUBTYPE_INDICATION



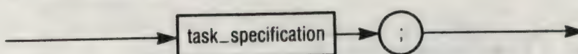
SUBUNIT



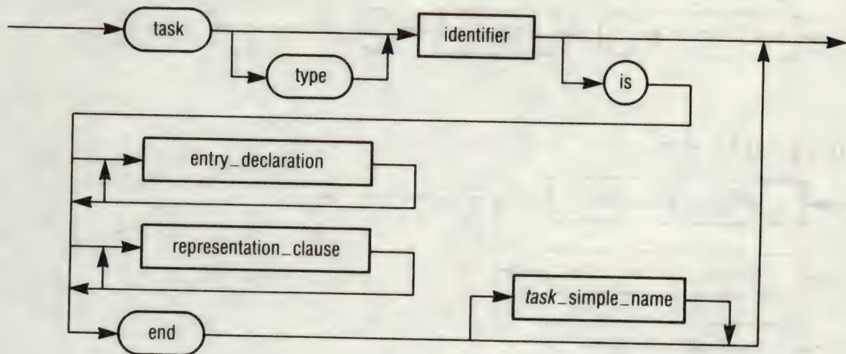
TASK_BODY



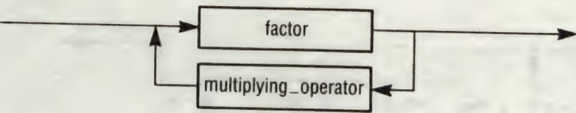
TASK_DECLARATION



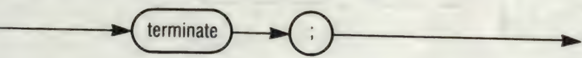
TASK_SPECIFICATION



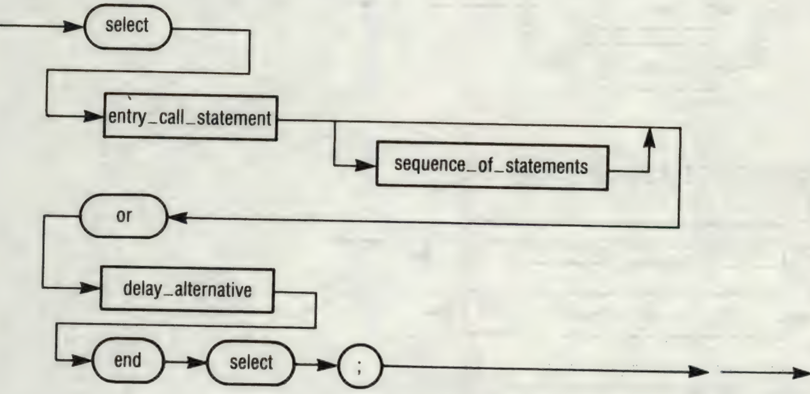
TERM



TERMINATE_ALTERNATIVE



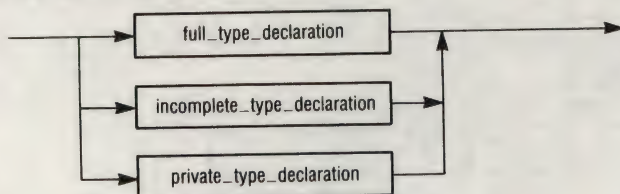
TIMED_ENTRY_CALL



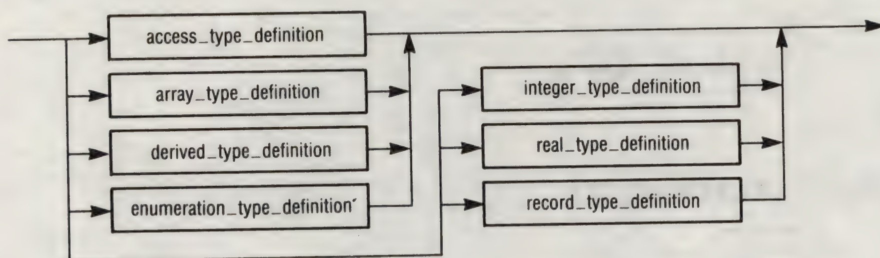
TYPE_CONVERSION



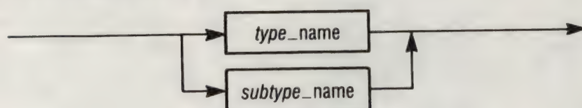
TYPE_DECLARATION



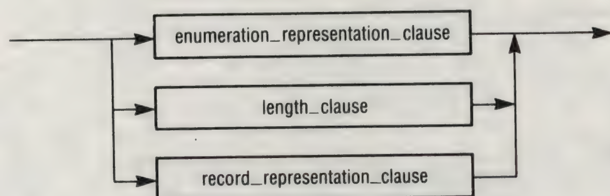
TYPE_DEFINITION



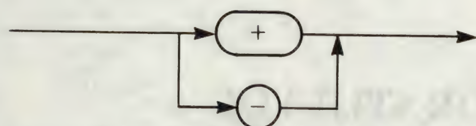
TYPE_MARK



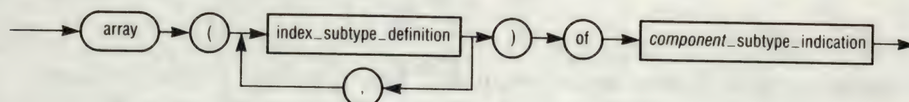
TYPE_REPRESENTATION_CLAUSE



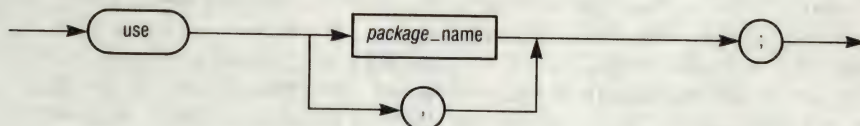
UNARY_ADDING_OPERATOR



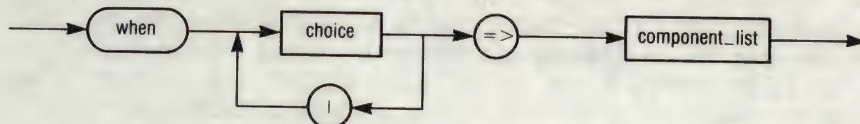
UNCONSTRAINED_ARRAY_DEFINITION



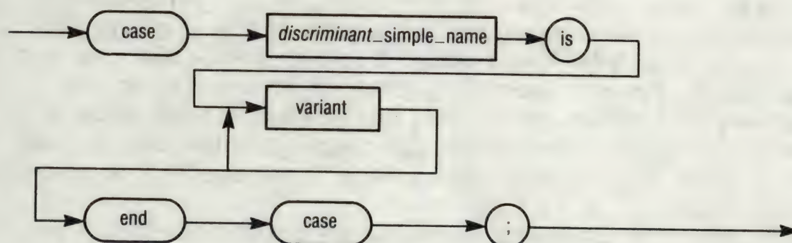
USE_CLAUSE



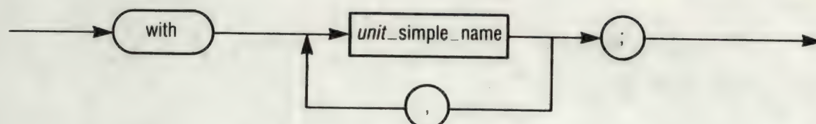
VARIANT



VARIANT_PART



WITH_CLAUSE



APPENDIX B: GIDS VOOR STIJLVOL PROGRAMMEREN IN ADA

Programmeerstijl is een onderwerp dat hevig wordt bediscussieerd, maar waarover het laatste woord zeker nog niet gesproken is. In ieder geval heeft de stijl waarin een systeem is ontwikkeld grote invloed op de begrijpelijkheid, onderhoudbaarheid en efficiëntie van dat systeem. In al onze voorbeelden hebben wij geprobeerd een stijl te handhaven, die deze doelstellingen bevordert en waarin tegelijkertijd van de mogelijkheden van Ada een goed gebruik wordt gemaakt. In deze appendix behandelen we programmeerstijl op drie niveaus en vatten we de in dit boek gedane aanbevelingen nog eens samen.

Als we het hebben over programmeerstijl denken velen aan kreten als: "Gij zult U onthouden van GOTO's" of "Iedere module dient hoogstens één pagina te beslaan". Dit zijn kunstmatige regels, die ertoe leiden dat de programmeur zich drukt maakt over details, terwijl hij de grote lijnen van het ontwerp uit het oog verliest. Als men er daarentegen naar streeft in zijn programmeerstijl de totale structuur te modelleren naar het in werkelijkheid waargenomen probleem, dan is de kans op een goed produkt veel groter.

Voor een dergelijke stijl kunnen hoogstens enkele richtlijnen worden gegeven. Ada is een omvangrijke taal, ontworpen voor het oplossen van complexe problemen en het is onmogelijk stijlregels te geven voor iedere mogelijke toepassing. Trouwens, als men aan een taal ook nog eens stijlregels toevoegt, moet de programmeur zich aan nog meer regels houden en bestaat het gevaar dat hij door de bomen het bos niet meer ziet en bepaalde mogelijkheden van de taal over het hoofd ziet, die tot een efficiënter of leesbaarder programma zouden kunnen leiden.

Voor de verdere behandeling verdelen we programmeerstijl in drie gedeelten:

- ontwerpstijl
- stijl bij het gebruik van de taal
- presentatiestijl

Ontwerpstijl

Een groep beroepsprogrammeurs werd ondervraagd over hun werkwijze bij het ontwerpen van een systeem. Ze moesten het antwoord schuldig blijven en na wat doorvragen werd duidelijk waarom: de programmeurs ontwierpen geen oplossingen, zij schreven gewoon programmacode.

Om betrouwbare oplossingen voor complexe problemen te kunnen ontwerpen moeten we consistente logische structuren ontwerpen. Een doelmatige structuur maakt de oplossing begrijpelijk en onderhoudbaar. Al gebruikt men ook de allerbeste gereedschappen voor het ontwikkelen van een oplossing, ook de krachtigste programmeertaal ter wereld kan van een slecht ontwerp geen goed ontwerp maken.

Zoals we al vaker vermeldden: we denken over een probleem in de werkelijkheid in termen van zelfstandige naamwoorden en werkwoorden. In Ada kunnen hiervoor parallellen worden gevonden in de vorm van objecten en operaties daarop. Om de verbinding te leggen tussen de werkelijkheid en de oplossing in de programmeertaal gebruikten we in dit boek de object-georiënteerde ontwerpmethode (zie hoofdstuk 5). Deze ontwerpstijl lijkt goed te werken voor een grote categorie van problemen en wordt door Ada uitstekend ondersteund.

De kern van deze ontwerpmethode is de mogelijkheid om gegevens en algoritmen op een bepaald abstractieniveau te behandelen en niet relevante implementatiedetails verborgen te houden (het Parnas decompositiecriterium). In tegenstelling tot zuivere functionele ontwerpmethodes erkent deze methode het belang van zowel objecten als operaties op objecten binnen de oplossing. We vatten dit samen in de volgende stijlregels:

- Gebruik een object-georiënteerde ontwerpstijl om zowel gegevens als besturingsstructuren op een voldoende hoog abstractieniveau te kunnen behandelen.
- Blijf u op elk niveau van abstractie bewust van de Hrair limiet, die aangeeft dat door de programmeur slechts een beperkt aantal grootheden tegelijkertijd overzien kan worden.
- Programma-eenheden dienen een hoge graad van cohesie en een lage graad van koppeling te bezitten (zie nog eens hoofdstuk 4).

Misschien bent u van mening dat een dergelijke stijl tot ondoelmatigheden moet leiden, vooral in het geval van real-time systemen, vanwege de overhead die het gevolg is van het creëren van een groot aantal verschillende eenheden. De overhead kan worden verminderd met behulp van de pragma INLINE, waarmee het doorgeven van parameters onnodig wordt, terwijl toch de leesbaarheid behouden blijft (zie hoofdstuk 20).

Een systeem dat een getrouw model is van de werkelijkheid is daardoor betrouwbaar; juist die onderdelen die geen afspiegeling

zijn van de realiteit vertonen het vaakst fouten. Verder geldt de 90-10 regel: in 90% van de gevallen wordt slechts van 10% van de code gebruik gemaakt. Wordt een bottleneck tijdens de verwerking geconstateerd, rafel dan niet het hele systeem uiteen, maar tracht die meest gebruikte 10% te identificeren en probeer die efficiënter te maken. Een goede ontwerpstijl zal tijdens de gehele levenscyclus een positieve uitwerking hebben; een slechte ontwerpstijl blijft u de hele levenscyclus achtervolgen!

Stijl Bij Het Gebruik Van De Taal

Niemand kan beweren dat hij alles weet van de Nederlandse taal, maar de meesten van ons weten er zich aardig in te redden. Meestal hebben we aan een kleine deelverzameling voldoende, hoewel in bepaalde gevallen een ingewikkelder constructie nodig kan zijn om een bepaalde betekenisnuance aan te geven. Dit geldt ook bij het gebruik van programmeertalen: meestal is een eenvoudige subset voldoende, maar in bepaalde gevallen moet een complexere structuur worden toegepast omdat deze het probleem genuanceerder definieert en/of efficiënter oplost. Bij het aanbevelen van een bepaalde stijl tijdens het gebruik van de taal zullen we het gebruik van bepaalde taalconstructies aanmoedigen en andere constructies weliswaar niet verbieden maar wel ten sterkste afraden.

Bij het geven van onze voorbeelden hebben we taalconstructies gesignaleerd die gevaarlijk bleken of alleen maar onduidelijk waren. Elke structuur kan op een goede en op een verkeerde manier worden gebruikt en omdat richtlijnen wat dit gebruik betreft te talrijk zijn om hier op te sommen, beperken wij ons tot enkele 'metarichtlijnen', dat wil zeggen richtlijnen voor richtlijnen. Gebruik de hier opgesomde mogelijkheden van Ada voor de volgende toepassingen:

- *Subprogramma's*
  Hoofdprogramma-eenheden
  Definitie van functies
  Definitie van operaties
- *Pakketten*
  Benoemde verzamelingen declaraties
  Groepen samenhangende programma-eenheden
  Abstracte datatypen
  Abstracte automaten
- *Taken*
  Parallel plaatsvindende acties
  Versturen van berichten
  Aansturen van randapparatuur
  Afvangen van interrupts

- *Generieke programma-eenheden*
  Algemeen formuleren van een klasse van programma-eenheden
  Doorgeven van typen als parameters naar programma-eenheden
- *Excepties*
  Afvangen van fouten
  Signaleren van en reageren op verwachte maar uitzonderlijke situaties

Oók aan de naamgeving van de gebruikte grootheden moet de nodige aandacht worden besteed. Zelfs simpele mogelijkheden in de taal, zoals het gebruik van het onderstrepingsteken binnen variabelenamen of het gebruik van benoemde parameters kan al sterk verduidelijkend werken. Hiervoor formuleren wij de volgende richtlijnen:

- Subprogramma's dienen werkwoordsvormen als namen te krijgen en functies die een boolean als resultaat opleveren moeten een vorm van het werkwoord 'zijn' bevatten:

  -- START_MENG_PROCES, SORTEER_LIJST, IS_NIET_LEEG

- Pakketten moeten benoemd worden met zelfstandige naamwoorden:

  -- WISKUNDIGE_FUNCTIES, SCHAAL_CONSTANTEN

- Taken worden eveneens met zelfstandige naamwoorden benoemd (meestal wordt een actie aangeduid):

  -- TIJDKLOK, BERICHTEN_REGELAAR, LIJST_DOORZOEKER

- Typen worden met gewone zelfstandige naamwoorden aangeduid:

  -- BOOM, VERBONDEN_LIJST, INDEX

- Objecten tenslotte krijgen eveneens zelfstandige naamwoorden als naam:

  -- MIJN_BOOM, VERBONDEN_LIJST_PERSONEEL, DATA_BASE_INDEX

Hoe u ook van een programmeertaal gebruik denkt te maken, houdt u in ieder geval altijd aan de volgende stelregel: als een keuze tussen twee of meer constructies mogelijk is, kies dan die met de helderste structuur.

Presentatiestijl

Presentatiestijl krijgt meestal de meeste aandacht; dit gebied is het makkelijkst te beschrijven en er kunnen hier het gemakkelijkst dwingende regels worden opgelegd. Regels voor stijl bij het gebruik van de programmeertaal bevelen het gebruik van bepaalde constructies aan, regels voor presentatiestijl daarentegen hebben betrekking op de fysieke opmaak of lay-out van de programmatekst. Het gaat bij de stijl van de presentatie louter om de lezer, op de werking van het programma heeft deze stijlvorm geen invloed.

De meeste stijlregels voor de presentatie houden zich bezig met zaken als aangeven van de programmastructuur door inspringen (indentatie), het gebruik van commentaar en de algehele programma lay-out. In het gehele boek hebben wij ons gehouden aan dezelfde indentatieregels en deze lijken dus voldoende geïllustreerd. Wat betreft interne documentatie via commentaren geldt voor de programmeertaal Ada, dat deze in belangrijke mate 'zelf-documenterend' is, door gebruik te maken van betekenisvolle namen en bijvoorbeeld benoemde parameters. Als verder van de mogelijkheid van afzonderlijke compilatie gebruik wordt gemaakt (en dit is aan te raden), dan behoeft slechts de lay-out van de afzonderlijke modulen verzorgd te worden; de problemen van het overzichtelijk maken van een lange lap tekst, zoals dat vaak nodig is bij andere programmeertalen, treden dan niet op.

De regels voor stijl van presentatie kunnen nu als volgt worden samengevat:

- Volg de aanbevolen stijl van inspringen ter weergave van de structuur.
- De overzichtelijkheid van de programmatekst kan worden bevorderd als met beleid van spatiëring en regels wit gebruik wordt gemaakt.
- Laat logisch samenhangende grootheden ook tesamen in hetzelfde programmatekstgedeelte voorkomen.
- Eenvoudige hulpmiddelen, zoals het netjes onder elkaar plaatsen van dubbele punten, of het op alfabet zetten van declaraties kunnen de leesbaarheid al sterk verbeteren.

Houdt in gedachten dat de programmatekst aangenaam moet ogen en dat die, als hij vervolgens nader wordt bekeken, bovendien begrijpelijk moet zijn. Bedenk dat een programma maar één maal geschreven wordt, maar talloze malen gelezen. Houdt tijdens het programmeren het gemak van de lezer voor ogen en niet dat van de schrijver van het programma!

APPENDIX C: DE VOORGEDEFINIEERDE TAALOMGEVING

De volgende appendix is met toestemming van het Ada Joint Program Office (OUSDRE), van het Departement van Defensie van de Verenigde Staten overgenomen uit: *Reference Manual for the Ada® Programming Language* (1983), Appendix C.

In de definitie van de programmeertaal Ada zijn een aantal voorgedefinieerde bibliotheekeenheden opgenomen. Het betreft onder andere:

- CALENDAR
- DIRECT_IO
- IO_EXCEPTIONS
- LOW_LEVEL_IO
- SEQUENTIAL_IO
- SYSTEM
- TEXT_IO
- UNCHECKED_CONVERSION
- UNCHECKED_DEALLOCATION

Verder is er een voorgedefinieerd pakket STANDARD, dat alle voorgedefinieerde objecten en operaties bevat. Als een bibliotheekeenheid wordt vertaald, dan wordt deze door de compiler behandeld alsof hij fysiek aan het eind van de specificatie van STANDARD voorkwam. Alle voorgedefinieerde grootheden zijn dan dus direct zichtbaar.

We geven hier de specificatie van het pakket STANDARD; de body is niet opgenomen omdat die implementatie-afhankelijk is. Het teken {...} geeft aan dat een waarde implementatie-afhankelijk is. Na STANDARD geven we de specificaties van de andere voorgedefinieerde programma-eenheden.

STANDARD

```
package STANDARD is
  --
  type BOOLEAN is (FALSE, TRUE);
  function "="   (LEFT , RIGHT : BOOLEAN) return BOOLEAN;
  function "/="  (LEFT , RIGHT : BOOLEAN) return BOOLEAN;
```

```
function "<"   (LEFT , RIGHT : BOOLEAN) return BOOLEAN;
function "<="  (LEFT , RIGHT : BOOLEAN) return BOOLEAN;
function ">"   (LEFT , RIGHT : BOOLEAN) return BOOLEAN;
function ">="  (LEFT , RIGHT : BOOLEAN) return BOOLEAN;
function "not" (X   : BOOLEAN) return BOOLEAN;
function "and" (X,Y : BOOLEAN) return BOOLEAN;
function "or"  (X,Y : BOOLEAN) return BOOLEAN;
function "xor" (X,Y : BOOLEAN) return BOOLEAN;
--
type INTEGER is {. . .};
function "="   (LEFT , RIGHT : INTEGER) return BOOLEAN;
function "/="  (LEFT , RIGHT : INTEGER) return BOOLEAN;
function "<"   (LEFT , RIGHT : INTEGER) return BOOLEAN;
function "<="  (LEFT , RIGHT : INTEGER) return BOOLEAN;
function ">"   (LEFT , RIGHT : INTEGER) return BOOLEAN;
function ">="  (LEFT , RIGHT : INTEGER) return BOOLEAN;
function "+"   (X   : INTEGER) return INTEGER;
function "-"   (X   : INTEGER) return INTEGER;
function "abs" (X   : INTEGER) return INTEGER;
function "+"   (X,Y : INTEGER) return INTEGER;
function "-"   (X,Y : INTEGER) return INTEGER;
function "*"   (X,Y : INTEGER) return INTEGER;
function "/"   (X,Y : INTEGER) return INTEGER;
function "rem" (X,Y : INTEGER) return INTEGER;
function "mod" (X,Y : INTEGER) return INTEGER;
function "**"  (LEFT , RIGHT : INTEGER) return INTEGER;
--
-- An implementation may provide additional predefined
-- integer types. It is recommended that the names of
-- such additional types end with INTEGER as in
-- SHORT_INTEGER or LONG_INTEGER
--
type FLOAT     is digits {. . .} range {. . .};
function "="   (LEFT , RIGHT : FLOAT) return BOOLEAN;
function "/="  (LEFT , RIGHT : FLOAT) return BOOLEAN;
function "<"   (LEFT , RIGHT : FLOAT) return BOOLEAN;
function "<="  (LEFT , RIGHT : FLOAT) return BOOLEAN;
function ">"   (LEFT , RIGHT : FLOAT) return BOOLEAN;
function ">="  (LEFT , RIGHT : FLOAT) return BOOLEAN;
function "+"   (X   : FLOAT) return FLOAT;
function "-"   (X   : FLOAT) return FLOAT;
function "abs" (X   : FLOAT) return FLOAT;
function "+"   (X,Y : FLOAT) return FLOAT;
function "-"   (X,Y : FLOAT) return FLOAT;
function "*"   (X,Y : FLOAT) return FLOAT;
function "/"   (X,Y : FLOAT) return FLOAT;
function "**"  (LEFT: FLOAT; RIGHT : INTEGER) return FLOAT;
--
-- An implementation may provide
-- additional floating point types.
```

```
-- It is recommended that the
-- names of such additional
-- types end with FLOAT, as in
-- SHORT_FLOAT or LONG_FLOAT;
--
-- The types universal-integer,
-- universal-float, and
-- universal-fixed are predefined.
--
function "*" (LEFT : universal-integer; RIGHT : universal-real) return universal-real;
function "*" (LEFT : universal-real; RIGHT : universal-integer) return universal-real;
function "/" (LEFT : universal-real; RIGHT : universal-integer) return universal-real;
function "*" (LEFT : any-fixed-type; RIGHT : any-fixed-type) return universal-fixed;
function "/" (LEFT : any-fixed-type; RIGHT : any-fixed-type) return universal-fixed;
--
-- The following characters comprise the standard ASCII character
-- set. Character literals corresponding to control characters are
-- not identifiers: They are underlined in this example.
--
type CHARACTER is (
   nul,  soh,  stx,  etx,  eot,  enq,  ack,  bel,
   bs,   ht,   lf,   vt,   ff,   cr,   so,   si,
   dle,  dc1,  dc2,  dc3,  dc4,  nak,  syn,  etb,
   can,  em,   sub,  esc,  fs,   gs,   rs,   us,
   ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
   '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
   '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
   '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
   '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
   'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
   'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
   'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
   '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
   'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
   'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
   'x',  'y',  'z',  '{',  '|',  '}',  '~',  del);
for CHARACTER use (0, 1, 2, . . . 126, 127);
package ASCII is
  -- control characters
  NUL  : constant CHARACTER := nul;
  SOH  : constant CHARACTER := soh;
  STX  : constant CHARACTER := stx;
  ETX  : constant CHARACTER := etx;
  EOT  : constant CHARACTER := eot;
  ENQ  : constant CHARACTER := enq;
  ACK  : constant CHARACTER := ack;
  BEL  : constant CHARACTER := bel;
  BS   : constant CHARACTER := bs;
```

```
HT   : constant CHARACTER := ht;
LF   : constant CHARACTER := lf;
VT   : constant CHARACTER := vt;
FF   : constant CHARACTER := ff;
CR   : constant CHARACTER := cr;
SO   : constant CHARACTER := so;
SI   : constant CHARACTER := si;
DLE  : constant CHARACTER := dle;
DC1  : constant CHARACTER := dc1;
DC2  : constant CHARACTER := dc2;
DC3  : constant CHARACTER := dc3;
DC4  : constant CHARACTER := dc4;
NAK  : constant CHARACTER := nak;
SYN  : constant CHARACTER := syn;
ETB  : constant CHARACTER := etb;
CAN  : constant CHARACTER := can;
EM   : constant CHARACTER := em;
SUB  : constant CHARACTER := sub;
ESC  : constant CHARACTER := esc;
FS   : constant CHARACTER := fs;
GS   : constant CHARACTER := gs;
RS   : constant CHARACTER := rs;
US   : constant CHARACTER := us;
DEL  : constant CHARACTER := del;
-- other characters
EXCLAM       : constant CHARACTER := '!';
SHARP        : constant CHARACTER := '#';
DOLLAR       : constant CHARACTER := '$';
QUERY        : constant CHARACTER := '?';
AT_SIGN      : constant CHARACTER := '@';
L_BRACKET    : constant CHARACTER := '[';
BACK_SLASH   : constant CHARACTER := '\';
R_BRACKET    : constant CHARACTER := ']'
CIRCUMFLEX   : constant CHARACTER := '^';
GRAVE        : constant CHARACTER := ' ' ';
L_BRACE      : constant CHARACTER := '{';
BAR          : constant CHARACTER := '|';
R_BRACE      : constant CHARACTER := '}';
TILDE        : constant CHARACTER := '~';
QUOTATION    : constant CHARACTER := '"';
COLON        : constant CHARACTER := ':';
SEMICOLON    : constant CHARACTER := ';';
PERCENT      : constant CHARACTER := '%'
AMPERSAND    : constant CHARACTER := '&';
UNDERLINE    : constant CHARACTER := '_';
-- lowercase letters
LC_A : constant CHARACTER := 'a';
. . .
LC_Z : constant CHARACTER := 'z';
end ASCII;
--
-- predefined types and subtypes
--
subtype NATURAL   is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE  is INTEGER range 1 .. INTEGER'LAST;
--
```

```
    type      STRING    is array (POSITIVE range <>) of CHARACTER;
    function "="  (LEFT , RIGHT : STRING) return BOOLEAN;
    function "/=" (LEFT , RIGHT : STRING) return BOOLEAN;
    function "<"  (LEFT , RIGHT : STRING) return BOOLEAN;
    function "<=" (LEFT , RIGHT : STRING) return BOOLEAN;
    function ">"  (LEFT , RIGHT : STRING) return BOOLEAN;
    function ">=" (LEFT , RIGHT : STRING) return BOOLEAN;
    function "&" (LEFT : STRING;    RIGHT : STRING)    return STRING;
    function "&" (LEFT : CHARACTER; RIGHT : STRING)    return STRING;
    function "&" (LEFT : STRING;    RIGHT : CHARACTER) return STRING;
    function "&" (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;
  pragma PACK(STRING);
  --
    type      DURATION is delta {. . .} range {. . .};
    --
    -- predefined exceptions
    --
    CONSTRAINT_ERROR : exception;
    NUMERIC_ERROR    : exception;
    PROGRAM_ERROR    : exception;
    STORAGE_ERROR    : exception;
    TASKING_ERROR    : exception;
end STANDARD;
```

CALENDAR

```
package CALENDAR is
  --
  type TIME is private;
  --
  subtype YEAR_NUMBER is INTEGER range 1901 . . 2099;
  subtype MONTH_NUMBER is INTEGER range 1 . .12;
  subtype DAY_NUMBER is INTEGER range 1 . . 31;
  subtype DAY_DURATION is DURATION range 0.0 . . 86_400.0;
  function CLOCK return TIME;
  --
  function YEAR    (T : TIME) return YEAR_NUMBER;
  function MONTH   (T : TIME) return MONTH_NUMBER;
  function DAY     (T : TIME) return DAY_NUMBER;
  function SECONDS (T : TIME) return DAY_DURATION;
  --
  procedure SPLIT   (DATE    : in TIME;
                     YEAR    : out YEAR_NUMBER;
                     MONTH   : out MONTH_NUMBER;
                     DAY     : out DAY_NUMBER;
                     SECONDS : out DAY_DURATION);
  function TIME_OF  (YEAR    : YEAR_NUMBER;
                     MONTH   : MONTH_NUMBER;
                     DAY     : DAY_NUMBER;
                     SECONDS : DAY_DURATION := 0.0)
            return TIME;
      TIME_ERROR : exception
  --
```

```
    function "+" (X : TIME;      Y : DURATION) return TIME;
    function "+" (X : DURATION; Y : TIME)      return TIME;
    function "-" (X : TIME;      Y : DURATION) return TIME;
    function "-" (X : TIME;      Y : TIME)     return DURATION;
    --
    function "<"  (X, Y : TIME) return BOOLEAN;
    function "<=" (X, Y : TIME) return BOOLEAN;
    function ">"  (X, Y : TIME) return BOOLEAN;
    function ">=" (X, Y : TIME) return BOOLEAN;
private
    -- implementation defined
end CALENDAR;
```

IO EXCEPTIONS

```
package IO_EXCEPTIONS is
    NAME_ERROR   : exception;
    USE_ERROR    : exception;
    STATUS_ERROR : exception;
    MODE_ERROR   : exception;
    DEVICE_ERROR : exception;
    END_ERROR    : exception;
    DATA_ERROR   : exception;
    LAYOUT_ERROR : exception;
end IO_EXCEPTIONS;
```

DIRECT IO

```
with IO_EXCEPTIONS;
generic
    type ELEMENT_TYPE is private;
package DIRECT_IO is
    --
    type FILE_TYPE is limited private;
    --
    type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
    type COUNT is range 0 . . implementation-defined;
    subtype POSITIVE_COUNT is COUNT range 1 . . COUNT'LAST;
    --
    procedure CREATE (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE := INOUT_FILE;
                      NAME  : in     STRING    := " ";
                      FORM  : in     STRING    := " ");
    procedure OPEN   (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE;
                      NAME  : in     STRING;
                      FORM  : in     STRING    := " ");
    --
    procedure CLOSE  (FILE  : in out FILE_TYPE);
```

```
procedure DELETE (FILE   : in out FILE_TYPE);
procedure RESET  (FILE   : in     FILE_TYPE;
                  MODE   : in     FILE_MODE);
procedure RESET  (FILE   : in     FILE_TYPE);
--
function MODE (FILE : in FILE_TYPE) return FILE_MODE;
function NAME (FILE : in FILE_TYPE) return STRING;
function FORM (FILE : in FILE_TYPE) return STRING;
--
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
--
procedure READ  (FILE   : in  FILE_TYPE;
                 ITEM   : out ELEMENT_TYPE);
procedure READ  (FILE   : in  FILE_TYPE;
                 ITEM   : out ELEMENT_TYPE;
                 FROM   : in  POSITIVE_COUNT);
procedure WRITE (FILE   : in  FILE_TYPE;
                 ITEM   : in  ELEMENT_TYPE);
procedure WRITE (FILE   : in  FILE_TYPE;
                 ITEM   : in  ELEMENT_TYPE;
                 TO     : in  POSITIVE_COUNT);
--
procedure SET_INDEX (FILE : in FILE_TYPE;
                     TO   : in POSITIVE_COUNT);
--
function INDEX       (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function SIZE        (FILE : in FILE_TYPE) return COUNT;
--
function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
--
NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
--
private
  -- implementation_defined
end DIRECT_IO;
```

LOW LEVEL IO

```
package LOW_LEVEL_IO is
  -- declaration of the possible types for DEVICE and DATA
  -- declaration of overloaded procedures for these types
  procedure SEND_CONTROL    (DEVICE :        device_type;
                             DATA   : in out data_type);
```

```
   procedure RECEIVE_CONTROL (DEVICE :         device_type;
                              DATA   : in out data_type);

end LOW_LEVEL_IO;
```

SEQUENTIAL IO

```
with IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
   --
   type FILE_TYPE is limited private;
   --
   type FILE_MODE is (IN_FILE, OUT_FILE);
   --
   procedure CREATE (FILE  : in out FILE_TYPE;
                     MODE  : in     FILE_MODE := OUT_FILE;
                     NAME  : in     STRING    := " ";
                     FORM  : in     STRING    := " ");
   procedure OPEN   (FILE  : in out FILE_TYPE;
                     MODE  : in     FILE_MODE;
                     NAME  : in     STRING;
                     FORM  : in     STRING    := " ");

   --
   procedure CLOSE  (FILE  : in out FILE_TYPE);
   procedure DELETE (FILE  : in out FILE_TYPE);
   procedure RESET  (FILE  : in     FILE_TYPE;
                     MODE  : in     FILE_MODE);
   procedure RESET  (FILE  : in     FILE_TYPE);
   --
   function MODE (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME (FILE : in FILE_TYPE) return STRING;
   function FORM (FILE : in FILE_TYPE) return STRING;
   --
   function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
   --
   procedure READ  (FILE : in  FILE_TYPE;
                    ITEM : out ELEMENT_TYPE);
   procedure WRITE (FILE : in  FILE_TYPE;
                    ITEM : in  ELEMENT_TYPE);
   --
   function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
   --
```

```
    NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
    STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
    DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
    --
private
    -- implementation defined
end SEQUENTIAL_IO;
```

SYSTEM

```
package SYSTEM is
    type ADDRESS is implementation_defined;
    type NAME    is implementation_defined;
    --
    SYSTEM_NAME  : constant NAME := implementation_defined;
    STORAGE_UNIT : constant      := implementation_defined;
    MEMORY_SIZE  : constant      := implementation_defined;
    MIN_INT      : constant      := implementation_defined;
    MAX_INT      : constant      := implementation_defined;
    MAX_DIGITS   : constant      := implementation_defined;
    MAX_MANTISSA : constant      := implementation_defined;
    FINE_DELTA   : constant      := implementation_defined;
    TICK         : constant      := implementation_defined;
    subtype PRIORITY is INTEGER range {. . .}
    . . .
end SYSTEM;
```

TEXT_IO

```
with IO_EXCEPTIONS;
package TEXT_IO is
    --
    type FILE_TYPE is limited private;
    --
    type FILE_MODE is (IN_FILE, OUT_FILE);
    --
```

```
subtype FIELD is INTEGER range 0 . . implementation-defined;
subtype NUMBER_BASE is INTEGER range 2 . . 16;
--
type COUNT is range 0 . . implementation-defined;
subtype POSITIVE_COUNT is COUNT range 1 . . COUNT'LAST;
type TYPE_SET is (LOWER_CASE, UPPER_CASE);
UNBOUNDED : constant COUNT := 0;
--
procedure CREATE (FILE   : in out FILE_TYPE;
                  MODE   : in     FILE_MODE  := OUT_FILE;
                  NAME   : in     STRING     := "";
                  FORM   : in     STRING     := "");
procedure OPEN    (FILE   : in out FILE_TYPE;
                  MODE   : in     FILE_MODE;
                  NAME   : in     STRING;
                  FORM   : in     STRING     := "");
--
procedure CLOSE  (FILE   : in out FILE_TYPE);
procedure DELETE (FILE   : in out FILE_TYPE);
procedure RESET  (FILE   : in     FILE_TYPE;
                  MODE   : in     FILE_MODE);
procedure RESET  (FILE   : in     FILE_TYPE);
--
function MODE (FILE : in FILE_TYPE) return FILE_MODE;
function NAME (FILE : in FILE_TYPE) return STRING;
function FORM (FILE : in FILE_TYPE) return STRING;
--
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
--
procedure SET_INPUT            (FILE : in FILE_TYPE);
procedure SET_OUTPUT           (FILE : in FILE_TYPE);
function  STANDARD_INPUT  return FILE_TYPE;
function  STANDARD_OUTPUT return FILE_TYPE;
function  CURRENT_INPUT   return FILE_TYPE;
function  CURRENT_OUTPUT  return FILE_TYPE;
--
procedure SET_LINE_LENGTH (FILE : in FILE_TYPE;
                           TO   : in COUNT);
procedure SET_LINE_LENGTH (TO   : in COUNT);
--
procedure SET_PAGE_LENGTH (FILE : in FILE_TYPE;
                           TO   : in COUNT);
procedure SET_PAGE_LENGTH (TO   : in COUNT);
--
function LINE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function LINE_LENGTH                       return COUNT;
--
function PAGE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function PAGE_LENGTH                       return COUNT;
--
```

```
procedure NEW_LINE (FILE       : in FILE_TYPE;
                    SPACING    : in POSITIVE_COUNT := 1);
procedure NEW_LINE (SPACING    : in POSITIVE_COUNT := 1);
--
procedure SKIP_LINE (FILE      : in FILE_TYPE;
                     SPACING   : in POSITIVE_COUNT := 1);
procedure SKIP_LINE (SPACING   : in POSITIVE_COUNT := 1);
--
function END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE                       return BOOLEAN;
--
procedure NEW_PAGE (FILE : in FILE_TYPE);
procedure NEW_PAGE;
--
procedure SKIP_PAGE (FILE : in FILE_TYPE);
procedure SKIP_PAGE;
--
function END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;
--
function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_FILE                       return BOOLEAN;
--
procedure SET_COL (FILE : in FILE_TYPE;
                   TO   : in POSITIVE_COUNT);
procedure SET_COL (TO   : in POSITIVE_COUNT);
--
procedure SET_LINE (FILE : in FILE_TYPE;
                    TO   : in POSITIVE_COUNT);
procedure SET_LINE (TO   : in POSITIVE_COUNT;
--
function COL (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function COL                       return POSITIVE_COUNT;
--
function LINE (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function LINE                       return POSITIVE_COUNT;
--
function PAGE (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function PAGE                       return POSITIVE_COUNT;
--
procedure GET (FILE : in  FILE_TYPE;
               ITEM : out CHARACTER);
procedure GET (ITEM : out CHARACTER);
procedure PUT (FILE : in  FILE_TYPE;
               ITEM : in  CHARACTER);
procedure PUT (ITEM : in  CHARACTER);
--
procedure GET (FILE : in  FILE_TYPE;
               ITEM : out STRING);
```

```
  procedure GET (ITEM  : out STRING);
  procedure PUT (FILE  : in  FILE_TYPE;
                 ITEM  : in  STRING);
  procedure PUT  (FILE : in  STRING);
  --
  procedure GET_LINE (FILE : in  FILE_TYPE;
                      ITEM : out STRING;
                      LAST : out NATURAL);
  procedure GET_LINE (ITEM : out STRING;
                      LAST : out NATURAL);
  procedure PUT_LINE (FILE : in  FILE_TYPE;
                      ITEM : in  STRING);
  procedure PUT_LINE (ITEM : in  STRING);
generic
  type NUM is range <>;
package INTEGER_IO is
  DEFAULT_WIDTH : FIELD := NUM'WIDTH;
  DEFAULT_BASE  : NUMBER_BASE := 10;
  procedure GET (FILE   : in  FILE_TYPE;
                 ITEM   : out NUM;
                 WIDTH  : in  FIELD := 0);
  procedure GET (ITEM   : out NUM;
                 WIDTH  : in  FIELD := 0);
  procedure PUT (FILE   : in  FILE_TYPE;
                 ITEM   : in  NUM;
                 WIDTH  : in  FIELD := DEFAULT_WIDTH;
                 BASE   : in  NUMBER_BASE := DEFAULT_BASE);
  procedure PUT (ITEM   : in  NUM;
                 WIDTH  : in  FIELD := DEFAULT_WIDTH;
                 BASE   : in  NUMBER_BASE := DEFAULT_BASE);
  procedure GET (FROM   : in  STRING;
                 ITEM   : out NUM;
                 LAST   : out POSITIVE);
  procedure PUT (TO     : out STRING;
                 ITEM   : in  NUM;
                 BASE   : in  NUMBER_BASE := DEFAULT_BASE);
end INTEGER_IO;
generic
  type NUM is digits <>;
package FLOAT_IO is
  DEFAULT_FORE : FIELD := 2;
  DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
  DEFAULT_EXP  : FIELD := 3;
  procedure GET (FILE   : in  FILE_TYPE;
                 ITEM   : out NUM;
                 WIDTH  : in  FIELD := 0);
  procedure GET (ITEM   : out NUM;
                 WIDTH  : in  FIELD := 0);
```

```
    procedure PUT (FILE  : in FILE_TYPE;
                   ITEM  : in NUM;
                   FORE  : in FIELD := DEFAULT_FORE;
                   AFT   : in FIELD := DEFAULT_AFT;
                   EXP   : in FIELD := DEFAULT_EXP);
    procedure PUT (ITEM  : in NUM;
                   FORE  : in FIELD := DEFAULT_FORE;
                   AFT   : in FIELD := DEFAULT_AFT;
                   EXP   : in FIELD := DEFAULT_EXP);
    --
    procedure GET (FROM  : in   STRING;
                   ITEM  : out NUM;
                   LAST  : out POSITIVE);
    procedure PUT (TO    : out STRING;
                   ITEM  : in   NUM;
                   AFT   : in   FIELD := DEFAULT_AFT;
                   EXP   : in   FIELD := DEFAULT_EXP);
end FLOAT_IO;
--
generic
  type NUM is delta <>;
package FIXED_IO is
  --
  DEFAULT_FORE : FIELD := NUM'FORE;
  DEFAULT_AFT  : FIELD := NUM'AFT;
  DEFAULT_EXP  : FIELD = 0;
  --
  procedure GET (FILE   : in   FILE_TYPE;
                 ITEM   : out NUM;
                 WIDTH  : in   FIELD := 0);
  procedure GET (ITEM   : out NUM;
                 WIDTH  : in   FIELD := 0;
  --
  procedure PUT (FILE  : in FILE_TYPE;
                 ITEM  : in NUM;
                 FORE  : in FIELD := DEFAULT_FORE;
                 AFT   : in FIELD := DEFAULT_AFT;
                 EXP   : in FIELD := DEFAULT_EXP);
  procedure PUT (ITEM  : in NUM;
                 FORE  : in FIELD := DEFAULT_FORE;
                 AFT   : in FIELD := DEFAULT_AFT;
                 EXP   : in FIELD := DEFAULT_EXP);
  --
  procedure GET (FROM  : in   STRING;
                 ITEM  : out NUM;
                 LAST  : out POSITIVE);
  procedure PUT (TO    : out STRING;
                 ITEM  : in   NUM;
```

```
                    AFT   : in  FIELD := DEFAULT_AFT;
                    EXP   : in  FIELD := DEFAULT_EXP);
end FIXED_IO;
--
generic
  type ENUM is (<>);
package ENUMERATION_IO is
  --
  DEFAULT_WIDTH : FIELD       := 0;
  DEFAULT_SETTING : TYPE_SET := UPPER_CASE;
  --
  procedure GET (FILE   : in  FILE_TYPE;
                 ITEM   : out ENUM);
  procedure GET (ITEM   : out ENUM);
  --
  procedure PUT (FILE   : in  FILE_TYPE;
                 ITEM   : in  ENUM;
                 WIDTH  : in  FIELD      := DEFAULT_WIDTH;
                 SET    : in TYPE_SET := DEFAULT_SETTING);
  procedure PUT (ITEM   : in  ENUM;
                 WIDTH  : in  FIELD      := DEFAULT_WIDTH;
                 SET    : in TYPE_SET := DEFAULT_SETTING);

  --
  procedure GET (FROM   : in  STRING;
                 ITEM   : out ENUM;
                 LAST   : out POSITIVE);
  procedure PUT (TO     : out STRING;
                 ITEM   : in  ENUM;
                 SET    : in TYPE_SET := DEFAULT_SETTING);
end ENUMERATION_IO;
--
  NAME_ERROR    : exception renames IO_EXCEPTIONS.NAME_ERROR;
  USE_ERROR     : exception renames IO_EXCEPTIONS.USE_ERROR;
  STATUS_ERROR  : exception renames IO_EXCEPTIONS.STATUS_ERROR;
  MODE_ERROR    : exception renames IO_EXCEPTIONS.MODE_ERROR;
  DEVICE_ ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
  END_ERROR     : exception renames IO_EXCEPTIONS.END_ERROR;
  DATA_ERROR    : exception renames IO_EXCEPTIONS.DATA_ERROR;
  LAYOUT_ERROR  : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;
private
  -- implementation_defined
end TEXT_IO;
```

UNCHECKED CONVERSION

```
generic
  type SOURCE is limited private ;
  type TARGET  is limited private ;
function UNCHECKED_CONVERSION(S : in SOURCE) return TARGET;
```

UNCHECKED DEALLOCATION

```
generic
  type OBJECT is limited private ;
  type NAME    is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X  :  in out NAME);
```

APPENDIX D: VOORGEDEFINIEERDE TAALATTRIBUTEN

De volgende appendix is met toestemming van het Ada Joint Program Office (OUSDRE) van het Departement van Defensie van de Verenigde Staten overgenomen uit: *Reference Manual for the Ada® Programming Language* (1983), Appendix A.

ATTRIBUUT	BETEKENIS
P'ADDRESS	*P is een object, een programma-eenheid, een label of een entiteit.* Levert het adres van de eerste aan P toegewezen geheugenlokatie. Voor subprogramma's, pakketten, taken of labels is dit het adres van de eerste machinecode-instructie van de bijbehorende body of instructie. Bij een taakentry met een adresclausule is het het adres van de hardware interrupt. De waarde van dit attribuut is van het type ADDRESS, zoals gedefinieerd in het pakket SYSTEM
P'AFT	*P is een fixed-point subtype.* Levert het aantal decimalen nodig voor benadering van de precisie van het subtype P. Als de delta van P echter groter is dan 0.1, dan is de waarde van P'AFT gelijk aan één. Het type van de waarde is 'universal integer'.
P'BASE	*P is een type of een subtype.* Geeft het basistype van P. Kan alleen gebruikt worden voor een ander attribuut, bijvoorbeeld: P'BASE'FIRST.
P'CALLABLE	*P moet geschikt zijn voor het type task.* Geef de waarde FALSE als de taak P gereed of beëindigd is, of als de taak op niet normale wijze onderbroken is. Levert anders de waarde TRUE op.

P'CONSTRAINED

P is een type met discriminanten.

Levert de waarde TRUE als een discriminant op P van toepassing is, of als het object een constante is (met inbegrip van een formele parameter of een generieke formele parameter van modus **in**); levert anders de waarde FALSE op. Is P een generieke formele parameter van modus **in out**, of een formele parameter van modus **in out**, en als het type in de bijbehorende parameterspecificatie verwijst naar een onbegrensd type met discriminanten, dan wordt de waarde van dit attribuut verkregen uit die van de overeenkomstige actuele parameter. De waarde van het attribuut is van het type BOOLEAN.

P'CONSTRAINED

P is een privaat type of een subtype.

Levert FALSE op als P een onbegrensd niet formeel privaat type met discriminanten is. Eveneens als P een generiek formeel privaat type is en het actuele subtype is, ofwel een onbegrensd type met discriminanten, ofwel een onbegrensd array type. Levert anders TRUE op. De waarde van het attribuut is van het type BOOLEAN.

P'COUNT

P is een taakeenheid.

Geeft het aantal entry-aanroepen in de wachtrij (als het attribuut wordt berekend binnen een accept-instructie voor entry P, dan wordt de aanroepende taak niet meegeteld). Waarde van het attribuut is van het type *universal integer*.

P'DELTA

P is een fixed point subtype.

Geeft de waarde van de delta, zoals gespecificeerd in de precisiedefinitie voor subtype P. Waarde van dit attribuut is van het type *universal real*.

P'DIGITS

P is een floating point subtype.

Levert het aantal decimale cijfers van de mantisse van getallen van het subtype P. Waarde van dit attribuut is van het type *universal integer*.

P'EMAX	*P is een floating point subtype.* Geeft de grootste waarde van de exponent van getallen van subtype P. Waarde van het attribuut is van het type *universal integer*.
P'EPSILON	*P is een floating point subtype.* Levert de absolute waarde van het verschil tussen het getal 1.0 en de direct volgende getalsvoorstelling van het subtype P. Waarde van het attribuut is van het type *universal real*.
P'FIRST	*P is een scalair subtype.* Geeft ondergrens van P. Waarde van het attribuut is van hetzelfde type als P.
P'FIRST	*P is geschikt als array type* Geeft de ondergrens van de eerste index van P. Waarde van het attribuut is van het-zelfde type als deze ondergrens.
P'FIRST(N)	*P is geschikt voor array type of verwijst naar een begrensd array subtype.* Geeft de ondergrens van de N-de index van P. N moet een statische integer zijn van het type *universal integer*. De waarde van N moet groter dan nul zijn en niet groter dan de dimensie van het array. De waarde van dit attribuut heeft hetzelfde type als de ondergrens.
P'FIRST_BIT	*P is een component van een record object.* Geeft de afstand vanaf het adres van de eerste geheugenlokatie voor deze component tot het eerste bit van de component. De afstand wordt gemeten in bits. De waarde van het attribuut is van type *universal integer*.
P'FORE	*P is een fixed point subtype* Geeft het minimale aantal tekens nodig voor het gehele deel van de decimale representatie van elke mogelijke waarde van het subtype P. Hierbij wordt er van uitgegaan, dat P geen exponent bevat, maar wel begint met één teken, namelijk een plus of een minteken of een spatie. Waarde van het attribuut is van het type *universal integer*.

P'IMAGE

P is een discreet type of subtype.

Dit attribuut is een functie met één parameter. De actuele parameter X moet een waarde hebben overeenkomend met het basistype van P. Het resultaat is van het voorgedefinieerde type STRING. Het resultaat is de voorstelling van de waarde van X, dat wil zeggen de rij tekens die de waarde van X representeert bij display. De voorstelling van een geheeltallige waarde is een rij decimale cijfers, zonder onderstrepingen, voorloopnullen, exponenttekens of afsluitende spaties, maar met een voorvoegsel van één teken: een plus, een min of een spatie.
De voorstelling van een enumeratiewaarde is ofwel de overeenkomstige naam in hoofdletters, ofwel het overeenkomstige letterteken (inclusief twee apostrofs), zonder spaties vooraf of achteraf. De afbeelding van niet grafische tekens is implementatie-afhankelijk.

P'LARGE

P is een real subtype.

Geeft de grootst voorstelbare waarde van subtype P. Waarde van dit attribuut is van type *universal real.*

P'LAST

P is een scalair subtype.

Geeft bovengrens van P. Waarde van dit attribuut heeft hetzelfde type als P.

P'LAST

P is geschikt als array type of verwijst naar een begrensd array subtype.

Geeft waarde bovengrens van eerste index van P. Waarde heeft hetzelfde type als deze bovengrens.

P'LAST(N)

P is geschikt als array type of verwijst naar een begrensd array subtype.

Geeft waarde van de bovengrens van de N-de index van P. N moet een statische expressie zijn van het type *universal integer*. N moet groter dan nul zijn, doch niet groter dan de dimensie van het array. Waarde van dit attribuut heeft hetzelfde type als de bovengrens.

P'LAST_BIT	*P is een component van een record object* Geeft afstand vanaf eerste geheugenlokatie van de component tot laatste bit bezet door deze component. De afstand wordt gemeten in bits. Waarde van attribuut is van type *universal integer.*
P'LENGTH	*P is geschikt als array type of verwijst naar een begrensd array subtype.* Geeft het aantal verschillende waarden van de eerste index van P. (Nul voor een leeg interval.) Waarde van attribuut is van type *universal integer.*
P'LENGTH(N)	*P is geschikt als array type of verwijst naar een begrensd array subtype.* Geeft aantal verschillende waarden van de N-de index van P. N moet een statische expressie zijn van het type *universal integer.* N moet groter dan nul zijn doch niet groter dan de dimensie van P. De waarde van het attribuut is van type *universal integer.*
P'MACHINE_EMAX	*P is een floating point type of een subtype.* Levert de grootst mogelijke waarde van de exponent in de machinevoorstelling van het basistype van P. Waarde van het attribuut is van type *universal integer.*
P'MACHINE_EMIN	*P is floating point type of subtype.* Geeft kleinste (meest negatieve) waarde van de exponent in de machinevoorstelling van het basistype van P. Waarde van attribuut is van type *universal integer.*
P'MACHINE_MANTISSA	*P is floating point type of subtype* Geeft aantal cijfers van mantisse van machinevoorstelling van basistype van P (deze cijfers lopen van 0 tot en met P'MACHINE_RADIX - 1). Waarde van het attribuut is van type *universal integer.*

P'MACHINE_OVERFLOWS

P is van type real of van subtype real

Geeft de waarde TRUE als elke voorgedefinieerde operatie op waarden van het basistype P ofwel tot een correct resultaat leidt, ofwel de exceptie NUMERIC_ERROR in overflow situaties genereert. Levert anders de waarde FALSE op. Waarde van attribuut is van type BOOLEAN.

P'MACHINE_RADIX

P is floating point type of subtype.

Geeft waarde van grondtal of radix van machinevoorstelling van het basistype van P. Waarde attribuut is van type *universal integer*.

P'MACHINE_ROUNDS

P is real type of subtype

Levert TRUE op als elke voorgedefinieerde rekenkundige bewerking op waarden van het basistype van P ofwel een exact resultaat oplevert, ofwel een afronding uitvoert. Levert anders FALSE op. Waarde van attribuut is van type BOOLEAN.

P'MANTISSA

P is een real subtype

Geeft aantal binaire cijfers in de binaire voorstelling van de mantisse van getallen van het subtype P. Waarde attribuut is van type *universal integer*.

P'POS

P is een discreet type of subtype.

Dit attribuut is een functie met één parameter. De actuele parameter X moet een waarde hebben van het basistype van P. Het resultaat is van het type *universal integer* en is het positienummer van de waarde van de actuele parameter.

P'POSITION

P is een component van een record object.

Geeft de afstand vanaf de eerste geheugenlokatie van het record tot aan de eerste lokatie van de component. De afstand wordt gemeten in opslageenheden. De waarde van het attribuut is van het type *universal integer*.

P'PRED

P is een discreet type of subtype.

Dit attribuut is een functie met één parameter. De actuele parameter moet een waarde hebben van het basistype van P. Deze waarde heeft een positienummer één kleiner dan dat van X. De exceptie CONSTAINT_ERROR ontstaat als X = P'BASE'FIRST.

P'RANGE

P moet geschikt zijn als array type of verwijzen naar een begrensd array subtype.

Levert het interval van de eerste index van P, dat wil zeggen P'FIRST .. P'LAST.

P'RANGE(N)

P moet geschikt zijn als array type of verwijzen naar een begrensd array subtype.

Levert het interval van de N-de index van P, dat wil zeggen P'FIRST(N) .. P'LAST(N).

P'SAFE_EMAX

P is een floating point type of subtype.

Geeft de grootste exponent in binaire vorm van toegelaten ('safe') getallen van het basistype van P. Waarde van het attribuut is van type *universal integer*.

P'SAFE_LARGE

P is een real type of subtype.

Levert het grootste positieve toegelaten getal van het basistype van P. Waarde van dit attribuut is van het type *universal real*.

P'SAFE_SMALL

P is een real type of subtype.

Levert het kleinste positieve toegelaten getal van het basistype van P. Waarde van dit attribuut is van het type *universal real*.

P'SIZE

P is een object.

Geeft het aantal bits dat voor het object is gereserveerd. Waarde van het attribuut is van type *universal integer*.

P'SIZE

P is van ieder type of subtype.

Levert het minimum aantal bits op, dat nodig is voor het opslaan van elk mogelijk object van subtype P. Waarde van het attribuut is van type *universal integer*.

P'SMALL

P is een real type of subtype.

Geeft het kleinste positieve voorstelbare getal van subtype P. Waarde van het attribuut is van type *universal real.*

P'STORAGE_SIZE

P is een access type of subtype.

Geeft het totale aantal opslageenheden dat gereserveerd is voor het basistype van P. Waarde van het attribuut is van type *universal integer.*

P'STORAGE_SIZE

P is een taaktype of taakobject.

Geeft het aantal opslageenheden gereserveerd voor elke activering van een taak van type P of voor de activering van de taak P. Waarde van het attribuut is van type *universal integer.*

P'SUCC

P is een discreet type of subtype.

Dit attribuut is een functie met één parameter. De actuele parameter moet een waarde hebben van het basistype van P. Het resultaat is eveneens van het basistype van P en stelt de waarde voor, waarvan het positienummer één groter is dan het positienummer van X. De exceptie CONSTRAINT_ERROR ontstaat als X = P'BASE'LAST.

P'TERMINATED

P is een taakobject.

Levert de waarde TRUE als de taak P beeindigd is en in alle andere gevallen de waarde FALSE. Waarde van het attribuut is van type BOOLEAN.

P'VAL

P is een discreet type of subtype

Dit is een speciale functie met één parameter die van elk geheeltallig type mag zijn. Het resultaat is van het basistype van P. Het is de waarde waarvan het positienummer overeenkomt met de actuele parameter. De exceptie CONSTRAINT_ERROR ontstaat als het resultaat niet ligt in het interval P'POS(P'FIRST)..(P'POS(P'LAST).

P'VALUE

P is een discreet type of subtype.

Dit attribuut is een functie met één parameter. De actuele parameter X moet een waarde zijn van het voorgedefinieerde type STRING. Het resultaat is van het basistype P. Voorlopende of afsluitende spaties in X worden genegeerd. Als P een enumeratietype is, dan is het resultaat de enumeratiewaarde. Als P een integer type is en de waarde van X, eventueel voorafgegaan door een plus- of een minteken, komt binnen het basistype van P voor, dan is het resultaat gelijk aan deze waarde. In alle andere gevallen ontstaat de exceptie CONSTRAINT_ERROR.

P'WIDTH

P is een discreet subtype.

Levert de maximale *image*-lengte over alle mogelijke waarden van het subtype van P. (De 'image' is de rij tekens die wordt geproduceerd via het attribuut IMAGE.) Waarde van dit attribuut is van het type *universal integer*.

APPENDIX E: VOORGEDEFINIEERDE TAALPRAGMA'S

De volgende appendix is met toestemming van het Ada Joint Program Office (OUSDRE) van het Departement van Defensie van de Verenigde Staten overgenomen uit: *Reference Manual for the Ada® Programming Language* (1983), Appendix B.

PRAGMA	BETEKENIS
CONTROLLED	Heeft als enige argument de niet samengestelde naam van een access type. Dit pragma mag alleen voorkomen binnen het declaratiegedeelte of de pakketspecificatie die de declaratie van het access type bevat. Deze declaratie moet vóór het pragma zelf in de tekst voorkomen. Voor een afgeleid type is dit pragma niet toegelaten. Het pragma geeft aan dat automatisch vrijmaken van niet meer gebruikte geheugenruimte voor objecten, waarnaar via het access type wordt verwezen, pas moet worden uitgevoerd bij het verlaten van het binnenste blok, de subprogrammabody, de taakbody waarbinnen de access type declaratie voorkomt, of na het verlaten van het hoofdprogramma.
ELABORATE	Heeft als argumenten één of meer niet samengestelde namen van bibliotheekeenheden. Mag alleen voorkomen onmiddellijk na een contextclausule van een compilatie-eenheid, en vóór de bibliotheekeenheid zelf. Elk argument moet een naam zijn van een bibliotheekeenheid, die in de contextclausule vermeld wordt. Dit pragma geeft aan dat de body van de bibliotheekeenheid moet worden uitgewerkt vóór de compilatie-eenheid. Als deze compilatie-eenheid een subeenheid is, dan moet de bibliotheekeenheidbody worden uitgewerkt vóór de bijbehorende hoofdbibliotheekeenheid van de subeenheid.

INLINE	Heeft als argumenten één of meer namen. Elke naam is de naam van een subprogramma of van een generiek subprogramma. Is alleen toegelaten op de plaats van een declaratie in een declaratiegedeelte of pakketspecificatie, of na een bibliotheekeenheid in een compilatie, maar vóór de volgende compilatie-eenheid. Dit pragma geeft aan dat de subprogrammabodies tijdens de compilatie binnen het aanroepende programma moeten worden uitgewerkt. In het geval van een generiek subprogramma heeft het pragma betrekking op aanroepen van de verschijningsvormen daarvan.
INTERFACE	Heeft de naam van een programmeertaal en de naam van een subprogramma als argumenten. Toegelaten op de plaats van een declaratie en moet betrekking hebben op een eerder gedeclareerd subprogramma. Het pragma is ook toegelaten voor een bibliotheekeenheid en moet dan staan na de subprogrammadeclaratie, maar voor eventuele volgende compilatie-eenheden. Het pragma specificeert een andere programmeertaal en de bijbehorende interfaceconventies en maakt aan de compiler duidelijk dat het aangegeven subprogramma in de vorm van objectcode zal worden aangeleverd.
LIST	Heeft de grootheden ON of OFF als enige argument. Is overal waar een pragma is toegelaten, toegelaten. Geeft aan dat het uitlijsten van de compilatiegegevens moet worden gestopt of juist gestart, totdat het tegengestelde pragma is aangegeven. Het pragma zelf wordt altijd afgedrukt als de compiler een uitlijsting produceert.
MEMORY_SIZE	Heeft een letterlijk voorgesteld getal als enige argument. Alleen toegelaten bij het begin van een compilatie, voor de eerste compilatie-eenheid. De waarde van het argument wordt gebruikt voor het getal MEMORY_SIZE.
OPTIMIZE	Heeft de grootheden TIME of SPACE als enige argument. Alleen toegelaten binnen een declaratiegedeelte en van toepassing op het blok of de body, waarin dit declaratiegedeelte voorkomt. Geeft aan of ofwel de verwerkingstijd ofwel de opslagruimte als belangrijkste optimalisatiecriterium door de compiler moet worden gekozen.

PACK	Heeft een niet samengestelde naam van een record of een array type als enige argument. Het pragma mag overal voorkomen waar ook een representatieclausule mag voorkomen. Geeft aan dat opslagruimteminimalisatie het belangrijkste criterium moet zijn bij de keuze van een voorstellingswijze van een bepaald type door de compiler.
PAGE	Dit pragma heeft geen argument en mag overal voorkomen waar een pragma mag voorkomen. Het geeft aan dat de programmatekst, die na dit pragma volgt, op een nieuwe pagina moet worden afgedrukt, als de compiler een uitlijsting produceert.
PRIORITY	Heeft een statische expressie van het voorgedefinieerde geheeltallige subtype PRIORITY als enige argument. Alleen toegelaten binnen de specificatie van een taakeenheid of onmiddellijk binnen het buitenste declaratiegedeelte van een hoofdprogramma. Specificeert de prioriteit van de taak (of van taken van dit taaktype) of de prioriteit van het hoofdprogramma.
SHARED	Heeft een niet samengestelde variabelenaam als enige argument. Alleen toegelaten voor een variabele die via een objectdeclaratie is gedeclareerd en van het scalaire of access type is. De variabele declaratie en het pragma moeten (in deze volgorde) onmiddellijk binnen hetzelfde declaratiegedeelte of dezelfde pakketspecificatie voorkomen. Het pragma geeft aan, dat het lezen en wijzigen van de variabele als synchronisatiepunt moet worden gebruikt. Een implementatie moet er zorg voor dragen dat dit pragma alleen is toegelaten voor objecten, waarvoor de lees- en wijzigoperatie als een ondeelbare operatie is geïmplementeerd.
STORAGE_UNIT	Heeft een letterlijk weergegeven getal als enige argument. Alleen toegelaten bij het begin van een compilatie, vóór de eerste compilatie-eenheid. Dit pragma gebruikt de waarde van het argument als waarde van het getal STORAGE_UNIT.

SUPPRESS

Heeft als argumenten de omschrijving van een controle en zonodig ook de naam van een object, een type of subtype, een subprogramma, een taakeenheid, of een generieke eenheid. Alleen toegelaten onmiddellijk binnen een declaratiegedeelte of onmiddellijk binnen een pakketspecificatie. In het laatste geval alleen toegelaten met een naam die naar een grootheid verwijst die meteen binnen de pakketspecificatie werd gedeclareerd. Toestemming tot het weglaten van de gespecificeerde controle loopt vanaf het pragma tot aan het einde van het declaratiegebied dat overeenkomt met het binnenste omsluitende blok of de binnenste omsluitende programma-eenheid. Wordt het pragma gegeven in een pakketspecificatie, dan strekt de toestemming zich uit tot aan het einde van het bereik van de benoemde grootheid.
Als het pragma een naam bevat dan wordt toestemming tot het weglaten van de aangegeven controle verder beperkt tot bewerkingen op het benoemde object of op alle objecten van het basistype van een benoemd type of subtype. Verder tot aanroepen van benoemde subprogramma's, tot activeringen van taken van het benoemde taaktype, of tot verschijningsvormen van de benoemde generieke eenheid.

SYSTEM_NAME

Heeft een letterlijk weergegeven enumeratiewaarde als enige argument. Kan alleen voorkomen bij het begin van een compilatie, vóór de eerste compilatie-eenheid. De waarde van het argument wordt de waarde van de constante SYSTEM_NAME. Dit pragma is alleen toegelaten als de waarde overeenkomt met het type NAME in het pakket SYSTEM.

APPENDIX F: OPLOSSINGEN VAN DE ONTWERPPROBLEMEN

Eerste Ontwerpprobleem: Het Tellen Van Blaadjes (Hoofdstuk 7)

```
pakket BLAADJES_PAKKET is
   type BLAADJES_TYPE is limited private;
   procedure TOON(BLAADJE: in      BLAADJES_TYPE);
   procedure TEL  (BLAADJE: in out BLAADJES_TYPE);
   procedure NUL  (BLAADJE: out    BLAADJES_TYPE);
private
   . . .
end BLAADJES_PAKKET;

package BOOM_PAKKET is
   type BOOM_TYPE is private;
   function IS_EEN_BLAADJE(BOOM    : in     BOOM_TYPE) return BOOLEAN;
   procedure SPLITS       (BOOM    : in out BOOM_TYPE;
                           LINKER  : out    BOOM_TYPE;
                           RECHTER : out    BOOM_TYPE);
   procedure GOOI_WEG     (BOOM    : in out BOOM_TYPE);
private
   . . .
end BOOM_PAKKET;

with BOOM_PAKKET;
package VERZAMELING_PAKKET is
   type VERZAMELING_TYPE is limited private;
   function  NIET_LEEG(VERZAMELING: in      VERZAMELING_TYPE) return BOOLEAN;
   procedure PLAATS   (BOOM        : in out BOOM_PAKKET.BOOM_TYPE;
                       IN          : in out VERZAMELING_TYPE);
   procedure PLAATS_OM_TE_BEGINNEN
                      (BOOM        : in out BOOM_PAKKET.BOOM_TYPE;
                       IN          : in out VERZAMELING_TYPE);
   procedure NEEM     (BOOM        : out    BOOM_PAKKET.BOOM_TYPE;
                       UIT         : in out VERZAMELING_TYPE);
private
   . . .
end VERZAMELING_PAKKET;
```

```
with BLAADJES_PAKKET,VERZAMELING_PAKKET,BOOM_PAKKET;
use  BLAADJES_PAKKET,VERZAMELING_PAKKET,BOOM_PAKKET;
procedure TEL_BLAADJES_IN_BINAIRE_BOOM is
  BLAADJES          : BLAADJES_TYPE;
  LINKER_SUBBOOM    : BOOM_TYPE;
  VERZAMELING       : VERZAMELING_TYPE;
  RECHTER_SUBBOOM   : BOOM_TYPE;
  BOOM              : BOOM_TYPE;
begin
  PLAATS_OM_TE_BEGINNEN(BOOM,IN => VERZAMELING);
  NUL(BLAADJES);
  while NIET_LEEG(VERZAMELING
    loop
      NEEM(BOOM,UIT => VERZAMELING);
      if IS_EEN_BLAADJE(BOOM) then
        TEL(BLAADJES);
        GOOI_WEG(BOOM);
      else
        SPLITS(BOOM,LINKER => LINKER_SUBBOOM,RECHTER => RECHTER_SUBBOOM);
        PLAATS(LINKER_SUBBOOM,IN => VERZAMELING);
        PLAATS(RECHTER_SUBBOOM,IN => VERZAMELING);
      end if;
    end loop;
    TOON(BLAADJES);
end TEL_BLAADJES_IN_BINAIRE_BOOM;
```

Het Tweede Ontwerpprobleem: Een Database Opvraagsysteem (Hoofdstukken 9 en 12)

```
package PROJECT is
  package EENHEID_INFORMATIE is
    type NAAM_TYPE              is new STRING(1 .. 20);
    type PROGRAMMEUR_TYPE       is new STRING(1 .. 20);
    type REFERENTIE;
    type REFERENTIE_TOEGANG     is access REFERENTIE;
    type REFERENTIE             is record
                                  DOCUMENT : STRING(1 .. 80);
                                  PAGINA   : POSITIVE;
                                  VOLGENDE : REFERENTIE_TOEGANG;
                                end record;
    type STATUS_TYPE            is (ONTWERP,CODE,TEST,OPERATIONEEL);
    type VERSIE_TYPE            is range 0 .. 99;
    type EENHEID_RECORD         is record
                                  PROGRAMMEUR  : PROGRAMMEUR_TYPE;
                                  SPECIFICATIE : REFERENTIE_TOEGANG;
                                  STATUS       : STATUS_TYPE;
                                  EENHEID_NAAM : NAAM_TYPE;
                                  VERSIE       : VERSIE_TYPE;
                                end record;
  end EENHEID_INFORMATIE;

  package DATA_BASE is
    use EENHEID_INFORMATIE;
    MAXIMUM_RECORDS        : constant := 100;
    type RECORD_INDEX      is range 0 .. MAXIMUM_RECORDS;
    type PROJECT_RECORDS   is array(RECORD_INDEX) of EENHEID_RECORD;
    ACTIEVE_RECORDS        : RECORD_INDEX;
    DATA                   : PROJECT_RECORDS;
  end DATA_BASE;
end PROJECT;
with SEQUENTIAL_IO;
package body PROJECT is
  package body DATA_BASE is
  package BASE_IO is new SEQUENTIAL_IO(EENHEID_RECORD);
  use BASE_IO
  DATA_FILE : BASE_IO.FILE_TYPE;
begin
  ACTIEVE_RECORDS := 0;
  OPEN(DATA_FILE,
       MODE => IN_FILE,
       NAME => "PROJECT_A/EENHEDEN");
  while not END_OF_FILE(DATA_FILE)
    loop
      ACTIEVE_RECORDS := ACTIEVE_RECORDS + 1;
      READ(DATA_FILE,ITEM => DATA(INDEX));
    end loop;
    CLOSE(DATA_FILE);
  end DATA_BASE;
end PROJECT;
```

```
with PROJECT;
use PROJECT;
package OPVRAAG_BEWERKINGEN is
  type COMMANDO is (VERZAMEL_KENTALLEN, PRODUCEER_LIJST,
                    STOP,               SELECTEER_EENHEID);
  function VERZOEK return OPDRACHT;
  procedure VERZAMEL_KENTALLEN;
  procedure PRODUCEER_LIJST;
  procedure SELECTEER_EENHEID;
end OPVRAAG_BEWERKINGEN;

separate (PRODUCEER_LIJST)
procedure SORTEER(RECORD   : in  DATA_BASE.PROJECT_RECORDS;
                  OMVANG   : in  DATA_BASE.RECORD_INDEX;
                  CRITERIA : in  SORTEERSLEUTEL;
                  LIJST    : out SORTEERLIJST) is
  TEMP_RECORD : EENHEID_INFORMATIE.EENHEID_RECORD;
  function NIET_OP_VOLGORDE(EERSTE   : in EENHEID_INFORMATIE.EENHEID_RECORD;
                            TWEEDE   : in EENHEID_INFORMATIE.EENHEID_RECORD;
                            CRITERIA : in SORTEERSLEUTEL) return BOOLEAN is
  begin
    case CRITERIA is
      when EENHEID      =>if EERSTE.EENHEID_NAAM> TWEEDE.EENHEID_NAAM then
                            return TRUE;
                          else
                            return FALSE;
                          and if;
      when SPECIFICATIE =>if EERSTE.SPECIFICATIE.DOCUMENT >
                            TWEEDE.SPECIFICATIE.DOCUMENT then
                            return TRUE;
                          else
                            return FALSE;
                          end if;
    end case;
  end NIET_OP_VOLGORDE;
begin
  for EERSTE_INDEX in 1 .. OMVANG - 1
    loop
      for TWEEDE_INDEX in EERSTE_INDEX + 1 .. OMVANG
        loop
          if NIET_OP_VOLGORDE(RECORDS(EERSTE_INDEX),
                              RECORDS(TWEEDE_INDEX),CRITERIA) then
            TEMP_RECORD           := RECORDS(EERSTE_INDEX);
            RECORDS(EERSTE_INDEX) := RECORDS(TWEEDE_INDEX);
            RECORDS(TWEEDE_INDEX) := TEMP_RECORD;
          end if;
        end loop;
    end loop;
  for INDEX in 1 .. OMVANG
    loop
      LIJST(INDEX) := INDEX;
    end loop;
end SORTEER;
```

```
with TEXT_IO;
use  TEXT_IO;
package body OPVRAAG_BEWERKINGEN is
  procedure PRINT(DATA_RECORD: in EENHEID_INFORMATIE.EENHEID_RECORD) is
    package STATUS_IO is new
              TEXT_IO.ENUMERATION_IO(EENHEID_INFORMATIE.STATUS_TYPE);
    package VERSIE_IO  is new
              TEXT_IO.INTEGER_IO(EENHEID_INFORMATIE.VERSIE_TYPE);
    use STATUS_IO,VERSIE_IO;
  begin
    SET_COL(TO => 1);
    PUT(DATA_RECORD.EENHEID_NAAM,WIDTH => 20);
    SET_COL(TO => 21);
    PUT(DATA_RECORD.PROGRAMMEUR, WIDTH => 20);
    SET_COL(TO => 42);
    PUT(DATA_RECORD.STATUS,WIDTH => 11);
    SET_COL(TO => 54);
    PUT(DATA_RECORD.VERSION,WIDTH => 5);
    SET_COL(TO => 62);
    PUT(DATA_RECORD.SPECIFICATIE.DOCUMENT,WIDTH => 80);
    NEW_LINE;
  end PRINT;

procedure PRINT_KOP is
  begin
    SET_COL(T= => 1);
    PUT("NAAM_EENHEID");
    SET_COL(TO => 21);
    PUT("NAAM_PROGRAMMEUR");
    SET_COL(TO => 42);
    PUT("STATUS");
    SET_COL(TO => 54);
    PUT("VERSIE");
    SET_COL(TO => 62);
    PUT("SPECIFICATIES");
    NEW_LINE
  end PRINT_KOP;

function VERZOEK return OPDRACHT is
  GEBRUIKERSOPVRAAG : OPDRACHT;
  package OPDRACHT_IO is new TEXT_IO.ENUMERATION_IO(OPDRACHT);
begin
  OPDRACHT_IO.GET(GEBRUIKERSOPVRAAG);
  TEXT_IO.NEW_LINE;
  return GEBRUIKERSOPVRAAG;
end VERZOEK;
```

```
procedure VERZAMEL_KENTALLEN is
  package STATUS_IO is new
            TEXT_IO.ENUMERATION_IO(EENHEID_INFORMATIE.STATUS_TYPE);
  use STATUS_IO;
  type ABSOLUUT is range 0 .. DATA_BASE.MAXIMUM_RECORDS;
  package ABSOLUUT_IO is new TEXT_IO.INTEGER_IO(ABSOLUUT);
  use ABSOLUUT_IO;
  type RELATIEF is delta 0.1 range 0.0 .. 100.0;
  package RELATIEF_IO is new TEXT_IO.FIXED_IO(RELATIEF);
  use RELATIEF_IO;
  type STATUS_RECORD is record
                          TOTAAL     : ABSOLUUT;
                          PERCENTAGE : RELATIEF;
                        end record;
  type STATUS_DATA is array (EENHEID_INFORMATIE.STATUS_TYPE)
                        of STATUS_RECORD;
  FREQUENTIE : STATUS_DATA := (ONTWERP .. OPERATIONEEL =>
                              (TOTAAL => 0, PERCENTAGE => 0.0));
begin
  PUT("KENTALLEN WORDEN VERZAMELD ...");
  NEW_LINE;
  PUT("STATUS       ABSOLUTE FREQUENTIE  RELATIEVE FREQUENTIE");
  NEW_LINE;
  for INDEX in 1 .. DATA_BASE.ACTIEVE_RECORDS
    loop
      FREQUENTIE(DATA_BASE.DATA(INDEX).STATUS).TOTAAL :=
               FREQUENTIE(DATA_BASE.DATA(INDEX).STATUS).TOTAAL + 1;
    end loop;
  for INDEX in EENHEID_INFORMATIE.STATUS_TYPE
    loop
      FREQUENTIE(INDEX).PERCENTAGE :=
               RELATIEF(FLOAT(FREQUENTIE(INDEX).TOTAAL))/
               DATA_BASE.ACTIEVE_RECORDS;
      PUT(INDEX,WIDTH => 11);
      PUT("       ");
      PUT(FREQUENTIE(INDEX).TOTAAL);
      PUT("                  ");
      PUT(FREQUENTIE(INDEX).PERCENTAGE);
      NEW_LINE;
    end loop;
end VERZAMEL_KENTALLEN;
```

```
procedure PRODUCEER_LIJST is
  type SORTEERSLEUTEL is (SPECIFICATIE,EENHEID);
  SORTEERCRITERIA : SORTEERSLEUTEL;
  package CRITERIA_IO is new TEXT_IO.ENUMERATION_IO(SORTEERSLEUTEL);
  use CRITERIA_IO;
  type SORTEERLIJST is array (DATA_BASE.RECORD_INDEX)
                   of DATA_BASE.RECORD_INDEX;
  SORTEERTABEL : SORTEERLIJST;
  procedure SORTEER(RECORDS  : in  DATA_BASE.PROJECT_RECORDS;
                    OMVANG   : in  DATA_BASE.RECORD_INDEX;
                    CRITERIA : in  SORTEERSLEUTEL;
                    LIJST    : out SORTEERLIJST) is separate;
  begin
    PUT("OP WELK VELD WILT U SORTEREN? ");
    GET(SORTEERCRITERIA);
    NEW_LINE;
    SORTEER(RECORDS  => DATA_BASE.DATA,
            OMVANG   => DATA_BASE.ACTIEVE_RECORDS,
            CRITERIA => SORTEERCRITERIA,
            LIJST    => SORTEERTABEL);
    PUT("SORTERING WORDT UITGEVOERD ...");
    NEW_LINE;
    PRINT_KOP;
    for INDEX in 1 .. DATA_BASE.ACTIEVE_RECORDS
      loop
        PRINT(DATA_BASE.DATA(SORTEERTABEL(INDEX)));
      end loop;
end PRODUCEER_LIJST;
procedure SELECTEER_EENHEID is
  EENHEID_NAAM : EENHEID_INFORMATIE.NAAM_TYPE;
begin
  PUT("OVER WELKE EENHEID WILT U GEGEVENS? ");
  GET(EENHEID_NAAM);
  NEW_LINE;
  PUT("SELECTIE VAN GEGEVENS VIA EENHEIDNAAM WORDT UITGEVOERD ...");
  NEW_LINE
  PRINT_KOP;
  for INDEX in 1 .. DATA_BASE.ACTIEVE_RECORDS
    loop
      if DATA_BASE.DATA(INDEX).EENHEID_NAAM = EENHEID_NAAM then
        PRINT(DATA_BASE.DATA(INDEX));
        exit;
      end if;
    end loop;
end SELECTEER_EENHEID;

end OPVRAAG_BEWERKINGEN;
```

```
with OPVRAAG_BEWERKINGEN,TEXT_IO;
use  OPVRAAG_BEWERKINGEN;
procedure  OPVRAGEN_VAN_PROJECTGEGEVENS is
begin
  loop
    TEXT_IO.PUT("VOER OPDRACHT IN: ");
    case OPVRAAG_BEWERKINGEN.VERZOEK is
      when VERZAMEL_KENTALLEN =>
           TEXT_IO.PUT("VERZAMEL_KENTALLEN COMMANDO GEACCEPTEERD");
           TEXT_IO;NEW_LINE;
           OPVRAAG_BEWERKINGEN.VERZAMEL_KENTALLEN;
      when PRODUCEER_LIJST       =>
           TEXT_IO.OUT("PRODUCEER_LIJST COMMANDO GEACCEPTEERD");
           TEXT_IO.NEW_LINE;
           OPVRAAG_BEWERKINGEN.PRODUCEER_LIJST;
      when STOP                  =>
           exit;
      when SELECTEER_EENHEID     =>
           TEXT_IO.PUT("SELECTEER_EENHEID COMMANDO GEACCEPTEERD");
           TEXT_IO;NEW_LINE;
           OPVRAAG_BEWERKINGEN.SELECTEER_EENHEID;
    end case;
  end loop;
end OPVRAGEN_VAN_PROJECTGEGEVENS;
```

Het Derde Ontwerpprobleem: Het Generieke pakket SET (Hoofdstuk 15)

```
generic
  type UNIVERSUM is (<>);
package SET_PAKKET is
  --
  type SET is private;
  NULL_SET : constant SET;
  --
  function "*"  (SET_1   : in SET       ;SET_2   : in SET)       return SET;
  function "+"  (ELEMENT : in UNIVERSUM;SET_1   : in SET)       return SET;
  function "+"  (SET_1   : in SET       ;SET_2   : in SET)       return SET;
  function "+"  (SET_1   : in SET       ;ELEMENT : in UNIVERSUM) return SET;
  function "-"  (SET_1   : in SET       ;SET_2   : in SET)       return SET;
  function "-"  (SET_1   : in SET       ;ELEMENT : in UNIVERSUM) return SET;
  function "<"  (SET_1   : in SET       ;SET_2   : in SET)       return BOOLEAN;
  function "<=" (SET_1   : in SET       ;SET_2   : in SET)       return BOOLEAN;
  --
  function IS_ELEMENT(ELEMENT : in UNIVERSUM;VAN_SET : in SET) return BOOLEAN;
  function IS_LEEG    (SET_1   : in SET)                       return BOOLEAN;
  subtype AANTAL is INTEGER range 0 .. (UNIVERSUM'POS(UNIVERSUM'LAST) -
                                 UNIVERSUM'POS(UNIVERSUM'FIRST)+1);
  function AANTAL_IN(SET_1 : in SET) return AANTAL;
private
  type SET is array (UNIVERSUM) of BOOLEAN;
  NULL_SET : constant SET := SET'(others => FALSE);
end SET_PAKKET;
package body SET_PAKKET is
  function "*"(SET_1 : in SET;SET_2 : in SET) return SET is
    -- doorsnede operator
  begin
    return (SET_1 and SET_2);
  end "*";
  function "+"(ELEMENT : in UNIVERSUM;SET_1 : in SET) return SET is
    -- vereniging operator
    WAARDE_VERZAMELING : SET := SET_1;
  begin
    WAARDE_VERZAMELING(ELEMENT) := TRUE;
    return WAARDE_VERZAMELING;
  end "+";
  function "+"(SET_1 : in SET;SET_2 : in SET) return SET is
    -- vereniging
  begin
    return (SET_1 or SET_2);
  end "+";
  function "+"(SET_1 : in SET;ELEMENT : in UNIVERSUM) return SET is
    -- vereniging
    WAARDE_VERZAMELING : SET := SET_1;
  begin
    WAARDE_VERZAMELING(ELEMENT) := TRUE;
    return WAARDE_VERZAMELING;
  end "+";
```

```
function "-"(SET_1 : in SET;ELEMENT : in UNIVERSUM) return SET is
  -- verschiloperator
  WAARDE_VERZAMELING: SET := SET_1;
begin
  WAARDE_VERZAMELING(ELEMENT) := FALSE;
  return WAARDE_VERZAMELING;
end "-";
function "-"(SET_1 : in SET;SET_2 : in SET) return SET is
  -- verschiloperator
begin
  return (SET_1 and (not SET_2));
end "-";
function "<="(SET_1 : in SET;SET_2 : in SET) return BOOLEAN is
  -- deelverzamelingsoperator
  WAARDE_VERZAMELING : SET := (SET_1 and SET_2);
begin
  return (WAARDE_VERZAMELING = SET_1);
end "<=";
function "<"(SET_1 : in SET;SET_2 : in SET) return BOOLEAN is
  -- echte deelverzamelingsoperator
  WAARDE_VERZAMELING : SET := (SET_1 and SET_2);
begin
  return((WAARDE_VERZAMELING = SET_1) and (WAARDE_VERZAMELING /= SET_2));
end "<";
function IS_ELEMENT(ELEMENT : in UNIVERSUM;VAN_SET : in SET)
                        return BOOLEAN is
  -- test op voorkomen element in verzameling
begin
  return VAN_SET(ELEMENT);
end IS_ELEMENT;
function IS_LEEG(SET_1 : in SET) return BOOLEAN is
  -- test op leeg zijn verzameling
begin
  return (SET_1 = NULL_SET);
end IS_LEEG;
function AANTAL_IN(SET_1 : in SET) return AANTAL is
  -- bepalen kardinaalgetal
  TEL : NATURAL := 0;
begin
  for INDEX in UNIVERSUM
    loop
      if SET_1(INDEX) then
        TEL := TEL + 1;
      end if;
    end loop;
  return TEL;
end AANTAL_IN;
end SET_PAKKET;
```

Vierde Ontwerpprobleem: Procesbesturing (Hoofdstuk 18)

```
with SYSTEM;
separate (BEWAAK_TEMPERATUREN)
task body ALARM is
  BITS        : constant := 1;
  WOORDEN : constant := 16*BITS;
  type LAMP        is (UIT,AAN);
  for  LAMP'SIZE use 1*WOORDEN;
  for  LAMP        use (UIT => 16#0000#,AAN => 16#FFFF#);
  DEFECT_LAMP   : LAMP := UIT;
  type  GRENSWAARDE_CONTROLE is array (VOELER_NAAM) of LAMP;
  for   GRENSWAARDE_CONTROLE'SIZE use
              (VOELER_NAAM'POS(VOELER_NAAM'LAST) + 1) * WOORDEN;
  GRENSWAARDE_LAMP : GRENSWAARDE_CONTROLE :=
                          GRENSWAARDE_CONTROLE'(others => UIT);
  for GRENSWAARDE_LAMP use at 16#0011#;
begin
  loop
    select
      accept VOELER_DEFECT do
        DEFECT_LAMP := AAN;
      end VOELER_DEFECT;
    or
      accept LIMIETWAARDEN_OVERSCHREDEN(OP_VOELER : in VOELER_NAAM) do
        GRENSWAARDE_LAMP(OP_VOELER) := AAN;
      end LIMIETWAARDEN_OVERSCHREDEN;
    end select;
  end loop;
end ALARM;

with APPARAAT_IO;
separate (BEWAAK_TEMPERATUREN)
task body REGISTRATIE_APPARAAT is
begin
  loop
    accept REGISTREER_STATUS(VAN_VOELER : in VOELER_NAAM;
                             MET_VOELER  : in VOELER_WAARDE;
                             MET_STATUS  : in VOELER_STATUS) do
      APPARAAT_IO.PUT(VAN_VOELER);
      APPARAAT_IO.PUT(MET_WAARDE);
      APPARAAT_IO.PUT(MET_STATUS);
    end REGISTREER_STATUS;
  end loop;
end REGISTRATIE_APPARAAT;
```

```
with SET_PAKKET,SYSTEEM;
separate (BEWAAK_TEMPERATUREN)
task body VOELERS is
  BITS      : constant := 1;
  WOORDEN : constant := 16*BITS;
  type VOELER_RECORD is record
    BOVENGRENS : VOELER_WAARDE := VOELER_WAARDE'LAST;
    ONDERGRENS : VOELER_WAARDE := VOELER_WAARDE'FIRST;
    WAARDE      : VOELER_WAARDE := VOELER_WAARDE'FIRST;
  end record;
  type VOELER_GROEP is array(VOELER_NAAM) of VOELER_RECORD;
  VOELER         : VOELER_GROEP;
  package  VOELER_SET is new SET_PAKKET(UNIVERSUM => VOELER_NAAM);
  use VOELER_SET;
  ACTIEVE_VOELERS : SET := NULL_SET;
  type VOELER_POORT is range  0 .. (2**WOORDEN - 1);
  for   VOELER_POORT'SIZE use 1*WOORDEN;
  type VOELER_LIJST is array(VOELER_NAAM) of VOELER_POORT;
  for   VOELER_LIJST'SIZE use
        (VOELER_NAAM'POS(VOELER_NAAM'LAST) + 1)*WOORDEN;
  VOELER_MAP : VOELER_LIJST;
  for   VOELER_MAP use at  16#0100#;
begin
  loop
    select
      accept SCHAKEL_UIT(VOELER : in  VOELER_NAAM) do
        ACTIEVE_VOELERS := ACTIEVE_VOELERS - VOELER;
      end SCHAKEL_UIT;
    or
      accept SCHAKEL_IN(VOELER : in  VOELER_NAAM) do
        ACTIEVE_VOELERS := ACTIEVE_VOELERS + VOELER;
      end SCHAKEL_IN;
    or
      accept REGISTREER_DIRECT(VAN_VOELER : in VOELER_NAAM) do
        if IS_ELEMENT(VAN_VOELER,VAN_SET => ACTIEVE_VOELERS) then
          REGISTRATIE_APPARAAT.REGISTEER_STATUS(VAN_VOELER,
                                     VOELER(VAN_VOELER).WAARDE,
                                     MET_STATUS => INGESCHAKELD);

        else
          REGISTRATIE_APPARAAT.REGISTREER_STATUS(VAN_VOELER,
                                     MET_WAARDE => VOELER_WAARDE'FIRST,
                                     MET_STATUS => UITGESCHAKELD);

        end if;
      end REGISTREER_DIRECT;
    or
      accept STEL_LIMIETWAARDEN_IN(VOOR_VOELER  : in  VOELER_NAAM;
                                   ONDERGRENS   : in  VOELER_WAARDE;
                                   BOVENGRENS   : in  VOELER_WAARDE) do
        VOELER(VOOR_VOELER).ONDERGRENS := ONDERGRENS;
        VOELER(VOOR_VOELER).BOVENGRENS := BOVENGRENS;
      end STEL_LIMIETWAARDEN_IN;
```

```
      else
        for I in VOELER_NAAM
          loop
            if IS_ELEMENT(I,VAN_SET => ACTIEVE_VOELERS) then
              VOELER(I).WAARDE := (VOELER_MAP(I)*VOELER_WAARDE(0.5));
              if (VOELER(I).WAARDE < VOELER(I).ONDERGRENS) or
                (VOELER(I).WAARDE > VOELER(I).BOVENGRENS) then
                ALARM.LIMIETWAARDEN_OVERSCHREDEN(I);
              end if;
            end if;
          end loop;
      end select;
    end loop;
end VOELERS;

separate (BEWAAK_TEMPERATUREN)
task body TIJDKLOK is
  MINUTEN : constant := 1;
  type INTERVAL is range 0 .. 15;
  TIKKEN : INTERVAL := 0;
begin
  loop
    accept INTERRUPT do
      TIKKEN := TIKKEN + 1;
      if TIKKEN = 15*MINUTEN then
        for I in VOELER_NAAM
          loop
            select
              VOELERS.REGISTREER_DIRECT(VAN_VOELER => I);
            or
              delay 5.0;
              ALARM.VOELER_DEFECT;
            end select;
          end loop;
        TIKKEN := 0;
      end if;
    end INTERRUPT;
  end loop;
end TIJDKLOK;
with TEXT_IO,SYSTEM;
use  TEXT_IO;
procedure BEWAAK_TEMPERATUREN is
  type COMMANDO       is (SCHAKEL_UIT,          SCHAKEL_IN,
                          REGISTREER_STATUS, STEL_LIMIETWAARDEN_IN);
  type VOELER_NAAM    is (HAL,             KANTOOR,            MAGAZIJN,
                          OPSLAGRUIMTE,    TERMINALKAMER,      BIBLIOTHEEK,
                          COMPUTERRUIMTE,ONTVANGSTRUIMTE,
                          LAADPLATFORM,    ,BEZEMKAST);
  type VOELER_STATUS is (UITGESCHAKELD,  INGESCHAKELD);
  type VOELER_WAARDE is delta 0.5 range 0.0 .. 100.0;
  package COMMANDO_IO      is new ENUMERATION_IO(COMMANDO);
  use     COMMANDO_IO;
  package VOELER_NAAM_IO   is new ENUMERATION_IO(VOELER_NAAM);
  use     VOELER_NAAM_IO;
  package VOELER_WAARDE_IO is new FIXED_IO(VOELER_WAARDE);
  use     VOELER_WAARDE_IO;
```

```
task ALARM is
  entry VOELER_DEFECT;
  entry LIMIETWAARDEN_OVERSCHREDEN(DOOR_VOELER : in VOELER_NAAM);
end ALARM;
task REGISTRATIE_APPARAAT is
  entry REGISTREER_STATUS(VAN_VOELER : in VOELER_NAAM;
                          MET_WAARDE : in VOELER_WAARDE;
                          MET_STATUS : in VOELER_STATUS);
end REGISTRATIE_APPARAAT;
task VOELERS is
  entry SCHAKEL_UIT             (VOELER      : in VOELER_NAAM);
  entry SCHAKEL_IN              (VOELER      : in VOELER_NAAM);
  entry TEGISTREER_DIRECT       (VAN_VOELER  : in VOELER_NAAM);
  entry STEL_LIMIETWAARDEN_IN   (VOOR_VOELER : in VOELER_NAAM;
                                 ONDERGRENS  : in VOELER_WAARDE;
                                 BOVENGRENS  : in VOELER_WAARDE);
end VOELERS;
task TIJDKLOK is
  entry INTERRUPT;
  for INTERRUPT use at 16#8E#;
end TIJDKLOK;
BOVENGRENS          : VOELER_WAARDE;
ONDERGRENS          : VOELER_WAARDE;
NAAM                : VOELER_NAAM;
GEBRUIKERSCOMMANDO  : COMMANDO;
WAARDE              : VOELER_WAARDE;

task body ALARM                 is separate;
task body REGISTRATIE_APPARAAT  is separate;
task body VOELERS               is separate;
task body TIJDKLOK              is separate;
--
begin
  loop
    begin  -- start van lokaal blok met exception handler
      PUT("Voer commando in: ");
      GET(GEBRUIKERSCOMMANDO);
      NEW_LINE;
      PUT_LINE("Commando geaccepteerd");
      case GEBRUIKERSCOMMANDO is
        when SCHAKEL_UIT =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          VOELERS.SCHAKEL_UIT(VOELER => NAAM);
          PUT_LINE("Voeler uitgeschakeld");
        when SCHAKEL_IN =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          VOELERS.SCHAKEL_IN(VOELER => NAAM);
          PUT_LINE("Voeler ingeschakeld");
        when REGISTREER_STATUS =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          VOELERS.REGISTREER_DIRECT(VAN_VOELER => NAAM);
          PUT_LINE("Voelerstatus geregistreerd");
```

```
        when STEL_LIMIETWAARDEN_IN =>
          PUT("Geef voelernaam: ");
          GET(NAAM);
          NEW_LINE;
          PUT("Geef ondergrens: ");
          GET(ONDERGRENS);
          NEW_LINE;
          PUT_LINE("Ondergrens geaccepteerd");
          PUT("Geef bovengrens: ");
          GET(BOVENGRENS);
          NEW_LINE;
          PUT_LINE("Bovengrens geaccepteerd");
          VOELERS.STEL_LIMIETWAARDEN_IN(VOOR_VOELER => NAAM,
                                        ONDERLIMIET => ONDERGRENS,
                                        BOVENLIMIET => BOVENGRENS);
          PUT_LINE("Limietwaarden ingesteld");
      end case;
      exception
        when DATA_ERROR =>
          PUT_LINE("Invoer niet correct, opnieuw alstublieft");
      end
  end loop;
end BEWAAK_TEMPERATUREN;
```

Het Vijfde Ontwerpprobleem: Het 'Kop-Op' Display (Hoofdstuk 21)

```
package INTERFACE_MET_HUD is
  type WAPEN_RECORD is ...
  procedure LEES_STATUS(VAN_WAPEN : out WAPEN_RECORD);
end INTERFACE_MET_HUD;
package INTERFACE_MET_HUD is
  type NAVIGATIE_RECORD is ...
  procedure LEES_STATUS(VAN_NAVIGATIE : out NAVIGATIE_RECORD);
end INTERFACE_MET_HUD;
package INTERFACE_MET_HUD is
  type TARGET_RECORD is ...
  procedure LEES_STATUS(VAN_TARGET : out TARGET_RECORD);
end INTERFACE_MET_HUD;

with WERELDSYSTEEM;
package TARGET is
  type TARGET_TYPE is new WERELDSYSTEEM.STATUSVECTOR;
  task DOEL is
    entry BEPAAL_WIJZIGINGEN(DOEL : out TARGET_TYPE);
  end DOEL;
end TARGET;

package WAPENSTATUS is
  type WAPEN is (VAST_BOORDWAPEN,RAKET,DOELZOEKEND_BOORDWAPEN);
  subtype HOEVEEL is INTEGER range 0 .. 1_000;
  type WAPEN_TYPE is
    record
      NAAM   : WAPEN;
      AANTAL : HOEVEELHEID;
    end record;
  task BEWAPENING is
    entry BEPAAL_WIJZIGINGEN(BEWAPENING : out WAPEN_TYPE);
  end BEWAPENING;
end WAPENSTATUS;

with WERELDSYSTEEM;
package VLUCHTPARAMETERS is
  type VLUCHT_TYPE is
    record
      HOOGTE       : WERELDSYSTEEM.AFSTAND;
      AANVALSHOEK  : WERELDSYSTEEM.RADIALEN;
    end record;
  task VLUCHT  is
    entry BEPAAL_WIJZIGINGEN(VLUCHT : out VLUCHT_TYPE);
  end VLUCHT;
end VLUCHTPARAMETERS;
```

```
with WERELDSYSTEEM;
package TARGETBOX is
  type RICHT_TYPE is
    record
      OP_TARGET : BOOLEAN;
      POSITIE   : WERELDSYSTEEM.COORDINAAT;
    end record;
  type BOX_TYPE is
    record
      MIDDEN  : WERELDSYSTEEM.COORDINAAT;
      OMVANG  : WERELDSYSTEEM.DIMENSIE;
      RICHT   : RICHT_TYPE;
    end record;
  task BOX is
    entry BEPAAL_WIJZIGINGEN(BOX : out BOX_TYPE);
  end BOX;
end TARGETBOX;

package GEBRUIKERSCOMMANDO is
  task WAARDE is
    entry IS_BEEINDIG;
  end WAARDE;
end GEBRUIKERSCOMMANDO;

with TARGET,    WAPENSTATUS,          VLUCHTPARAMETERS,
     TARGETBOX, GEBRUIKERSCOMMANDO;
use  TARGET,    WAPENSTATUS,          VLUCHTPARAMETERS,
     TARGETBOX, GEBRUIKERSCOMMANDO;
procedure HEADS_UP_DISPLAY is
  WAPEN_DATA  : WAPEN_TYPE;
  BOX_DATA    : BOX_TYPE;
  VLUCHT_DATA : VLUCHT_TYPE;
  TARGET_DATA : TARGET_TYPE;
  procedure PAS_AAN_BEWAPENING(A : in WAPEN_TYPE)  is separate;
  procedure PAS_AAN_BOX       (B : in BOX_TYPE)    is separate;
  procedure PAS_AAN_VLUCHT    (V : in VLUCHT_TYPE) is separate;
  procedure PAS_AAN_DOEL      (T : in TARGET_TYPE) is separate;
begin
  loop
    select
      DOEL.BEPAAL_WIJZIGINGEN(TARGET_DATA);
      PAS_AAN_TARGET(TARGET_DATA);
    else
      null;
    end select;
    select
      BEWAPENING.BEPAAL_WIJZIGINGEN(WAPEN_DATA);
      PAS_AAN_BEWAPENING(WAPEN_DATA);
    else
      null;
    end select;
    select
      GEBRUIKERSCOMMANDO.WAARDE.IS_BEEINDIG;
      exit;
    else
      null;
    end select
  end loop;
end HEADS_UP_DISPLAY;
```

APPENDIX G: UITWERKINGEN VAN ENIGE OEFENINGEN

Hoofdstuk 2

3. Deze uitspraak is waar. Onderzoekingen hebben uitgewezen dat het corrigeren van een fout in de testfase wel 20 maal zoveel kan kosten als de correctie van dezelfde fout in de programmeerfase. Verder geldt: hoe vroeger een fout in het ontwerpproces wordt gemaakt, deste later wordt hij vaak ontdekt. Programmeerfouten worden veelal vaak bij het in gebruik nemen van het systeem ontdekt.

Hoofdstuk 3

3. Bij het ontwerpen van Ada is gedacht aan een specifiek probleemgebied: 'embedded systems'. Voor technische berekeningen werd en wordt FORTRAN gebruikt, voor administratieve toepassingen meestal COBOL, maar dit wil niet zeggen dat Ada ook niet voor deze toepassingsgebieden geschikt is.

Hoofdstuk 4

1. Neem bijvoorbeeld dit boek. Het kan op de volgende abstractieniveaus bekeken worden:

- atomen
- papier
- symbolen
- woorden
- zinnen
- alineas
- paragrafen
- hoofdstukken
- onderwerpen
- ideeën

Hoofdstuk 5

2. Ja, ook in een assembleertaal kunnen abstracte begrippen worden gerealiseerd. Assembleertalen hebben echter niet al te veel uitdrukkingsmogelijkheid en het creëren van een leesbare en begrijpelijke abstractie is dus niet eenvoudig. Verder is van automatisch dwingend opleggen van het correcte gebruik van eenmaal gecreëerde abstracties in assembleertalen al helemaal geen sprake.

Hoofdstuk 6

2. Als we ons beperken tot het hoogste abstractieniveau, dan worden de volgende principes voor goed software-ontwerp door de volgende Ada eigenschappen ondersteund:

- *Abstractie en information hiding*

  Datatype mechanismen.
  De onderverdeling van elke programma-eenheid in een specificatiegedeelte en een implementatiegedeelte.
  Het pakketmechanisme voor abstracte datastructuren.
  Subprogramma's en taken voor abstracte besturingsstructuren.

- *Modulariteit en lokalisatie*

  Afzonderlijke compileerbaarheid.
  De drie klassen van programma-eemheden: subprogramma's, pakketten en taken.
  Bibliotheekeenheden voor bottom-up ontwikkeling.
  Subeenheden voor top-down ontwikkeling.

- *Uniformiteit, volledigheid en testbaarheid*

  Datatype mechanismen.
  Controle en correct gebruik van typen (type checking) ook over afzonderlijk gecompileerde eenheden.
  Consistentie van de structuur van de taalsyntaxis.

Deze lijst pretendeert volstrekt niet volledig te zijn.

Hoofdstuk 7

1. De volgende oplossing maakt gebruik van een recursieve formulering. (Het weergeven van de telling lieten we eenvoudigheidshalve weg.)

```
with BOOM_PAKKET;
use  BOOM_PAKKET;
function BLAADJES_TELLEN(T : in BOOM) return INTEGER is
  LINKER_SUBBOOM   : BOOM;
  RECHTER_SUBBOOM : BOOM;
begin
  if IS_EEN_BLAADJE(T) then
    return 1;
  else
    SPLITS(T,LINKER_SUBBOOM,RECHTER_SUBBOOM);
    return BLAADJES_TELLEN(LINKER_SUBBOOM) +
           BLAADJES_TELLEN(RECHTER_SUBBOOM);
  end if;
end BLAADJES_TELLEN;
```

3. De onderlinge afhankelijkheid tussen programma-eenheden kan worden vastgesteld via de with-clausules; deze geven de context aan, die voor de programma-eenheid bekend moet zijn. Zouden we hier de voorstellingswijze van BOOM_TYPE wijzigen, dan heeft dat alleen gevolgen voor VERZAMELING_PAKKET en voor TEL_BLAADJES_IN_BINAIRE_BOOM; deze twee eenheden moeten opnieuw worden gecompileerd. Logisch gezien verandert er niets, omdat we alleen van bepaalde zichtbare faciliteiten van BOOM_TYPE (namelijk de operaties) gebruik maken. Er is dus sprake van invariantie op logisch niveau bij een wijziging op fysiek niveau.

Hoofdstuk 8

3. (a) type TELLER range INTEGER'FIRST .. -1;
   (b) type COEFFICIENT digits 12;
   (c) type METING delta 1.0 range FLOAT'FIRST .. FLOAT'LAST;
   (d) type METING delta 1.0 range 0.0 .. 1000.0;
   (e) type KLEUREN_VAN_DE_REGENBOOG is (ROOD,ORANJE,GEEL, GROEN, INDIGO,VIOLET);

5. (a) type MIJN_ARRAY is range (INTEGER range 1 .. 10) of COEFFICIENT;
   (b) type WAARNEMING is array (KLEUREN_VAN_DE_REGENBOOG range <>) of METING;
   (c) type WAARNEMING is array (KLEUREN_VAN_DE_REGENBOOG range ROOD .. GEEL) of METING;
   (d)

```
type PERSOONSRECORD is
  record
    LEEFTIJD : NATURAL;
    GEWICHT  : NATURAL;
    LENGTE   : FLOAT;
    NAAM     : STRING(1 .. 20);
  end record;
```

   (e)

```
type GENERATIE is (KIND,VOLWASSENE);
type PERSOONSRECORD(KLASSE : GENERATIE) is
  record
    LEEFTIJD : NATURAL;
    GEWICHT  : NATURAL;
    LENGTE   : FLOAT;
    NAAM     : STRING(1 .. 20);
    case KLASSE is
      when KIND =>
        NAAM_MOEDER : STRING(1 .. 20);
        NAAM_VADER  : STRING(1 .. 20);
      when VOLWASSENE =>
        BEROEP      : STRING(1 .. 30);
    end case;
  end record;
```

6. type PERSOONSPOINTER is access PERSOONSRECORD;

7. Nu is een onvolledige typedeclaratie nodig:

```
type PERSOONSRECORD(KLASSE : GENERATIE);
type PERSOONSPOINTER is access PERSOONSRECORD;
type PERSOONSRECORD(KLASSE : GENERATIE) is
  record
    LEEFTIJD : NATURAL
    GEWICHT  : NATURAL;
    LENGTE   : FLOAT;
    NAAM     : STRING(1 .. 20);
    case KLASSE is
      when KIND =>
        NAAM_MOEDER : PERSOONSPOINTER;
        NAAM_VADER  : PERSOONSPOINTER;
      when VOLWASSENE =>
        BEROEP      : STRING(1 .. 30);
    end case;
  end record;
```

```
8.  ANNA    : PERSOONSPOINTER := new PERSOONSRECORD(KIND);
    LIDA    : PERSOONSPOINTER := new PERSOONSRECORD(VOLWASSENE);
    MAARTEN : PERSOONSPOINTER := new PERSOONSRECORD(VOLWASSENE);
    . . .
    ANNA.NAAM_MOEDER := LIDA;
    ANNA.NAAM_VADER  := MAARTEN;

10. (a) subtype KLEINE_PRECISIE is COEFFICIENT digits 7;
    (b) subtype POSITIEF_KLEIN  is KLEINE_PRECISIE
          range  0.0 .. KLEINE_PRECISIE'LAST;
    (c) type AFGELEID_KLEIN is new KLEINE_PRECISIE
          range  0.0 .. KLEINE_PRECISIE'LAST;
    (d) subtype  KINDRECORD is PERSOONSRECORD(KLASSE => KIND);
    (e) type AFGELEID_KIND  is new PERSOONSRECORD(KLASSE => KIND);

11. (a) X_FACTOR : KLEINE_PRECISIE := -3.141_592;
    (b) Y_FACTOR : POSITIEF_KLEIN  :=  2.789_532_678;
                                   -- getal wordt geconverteerd naar toegelaten
                                   -- getal met minstens 7 cijfers
    (c) Z_FACTOR : AFGELEID_KLEIN  := AFGELEID_KLEIN(Y_FACTOR);
                                   -- hier wordt typeconversie toegepast
    (d) BABY : KIND_RECORD := KIND_RECORD(KIND,
           LEEFTIJD      => 2;
           GEWICHT       => 45.0;
           LENGTE        => 2.5;
           NAAM          => "MASJA";
           NAAM_MOEDER   => "LIDA";
           NAAM_VADER    => "MAARTEN");
    (e) NOG_EEN_KIND : AFGELEID_KIND := AFGELEID_KIND(BABY);
```

Hoofdstuk 9

```
1.  with PROJECT;
    use  PROJECT;
    package SORTEER_BEWERKINGEN is
      type COMMANDO is (OPLOPEND,AFLOPEND);
      procedure SORTEER_OPLOPEND;
      procedure SORTEER_AFLOPEND;
    end SORTEER_BEWERKINGEN;
```

3. Bij directe toegang tot de DATA_BASE zou de gebruiker tegenstrijdigheden in de gegevens kunnen introduceren en de beveiliging die het pakketmechanisme biedt te niet kunnen doen. Ook hier wordt de fysieke representatie voor de gebruiker verborgen gehouden om de abstracte representatie te kunnen waarborgen. Verder moet worden opgemerkt dat, als de gebruiker de fysieke representatie zou kennen, het gevaar bestaat dat hij bij zijn applicaties van deze kennis gebruik maakt. Zou nu de fysieke (maar niet de logische) voorstellingswijze worden veranderd, dan zouden ook alle gebruikersapplicaties moeten worden aangepast.

4. We gebruikten hier parameterloze procedures, omdat de gebruiker de database niet direct hoeft te kunnen zien. De gebruiker kan het pakket zien als een automaat met de subprogramma's als operaties.

Hoofdstuk 10

1. **procedure** NORMALISEER(GETAL : **in out** FLOAT);

2. **function** MATRIX_VERMENIGVULDIGING(M_1,M_2 : **in** MATRIX) **return** MATRIX;
function "*"(M_1,M_2 : **in out** MATRIX) **return** MATRIX);

3. **procedure** BESTEL_DINER
```
                (VOORGERECHT  : VOORGERECHT_TYPES := GARNALENCOCKTAIL;
                (HOOFDGERECHT : HOOFDSCHOTELS     := KALFSOESTER;
                 GROENTE      : GROENTE_SOORTEN   := SNIJBONEN;
                 DRANK        : DRANKEN           := BORDEAUX_WIJN;
                 DESSERT      : DESSERT_TYPES     := VRUCHTENIJS);
```

4. **function** IS_LEEG(STACK : STACK_TYPE) **return** BOOLEAN;

Hoofdstuk 11

1.
```
type ZIJDE is range 1 .. 10;
type DRIE_D_MATRIX is array (ZIJDE,ZIJDE,ZIJDE) of FLOAT;
KUBUS : DRIE_D_MATRIX;
. . .
```
De hoeken zijn: KUBUS(1,1,1) KUBUS(1,1,10) KUBUS(1,10,1)
KUBUS(1,10,10) KUBUS(10,1,1) KUBUS(10,1,10)
KUBUS(10,10,1) KUBUS(10,10,10)

2.
```
type CODE is record
                 CARRY      : BOOLEAN;
                 NUL        : BOOLEAN;
                 NEGATIEF   : BOOLEAN;
                 PRIORITEIT : INTEGER range 0 .. 31;
             end record;
CONDITIE_CODE : CODE;
. . .
```
De component heet CONDITIE_CODE.PRIORITEIT

3.
```
type CODE_POINTER is access CODE;
STATUS_WOORD : CODE_POINTER := new CODE;
STATUS_WOORD.all := CONDITIE_CODE;
```

Het access-object heet STATUS_WOORD. STATUS_WOORD.all is de naam die verwijst naar het gehele toegewezen object. De component NUL is STATUS_WOORD.NUL.

4.
```
KUBUS := DRIE_D_MATRIX'(1 =>      (others =>
                                  (others => 1.0)),
                        others => (others =>
                                  (others => -1.0)));
```

5. De volgende twee vormen zijn gelijkwaardig, maar de eerste wordt vanwege de betere leesbaarheid geprefereerd:

```
CONDITIE_CODE := CODE'(CARRY|NULL => FALSE,
                       NEGATIEF   => TRUE,
                       PRIORITEIT => 7);
CONDITIE_CODE := (FALSE,FALSE,TRUE,7);
```

Hoofdstuk 12

1. Door een niet toegelaten commando zou de exceptie DATA_ERROR ontstaan in de functie VERZOEK. Deze functie kent geen exception handler, dus wordt de fout naar buiten gepropageerd naar de module OPVRAGEN_VAN_PROJECT_GEGEVENS. Als het hoofdprogramma wel een exception handler bezit, dan vangt deze de exceptie op. Is er geen exception handler dan wordt de besturing teruggegeven aan de programmeeromgeving.

Hoofdstuk 13

```
package COMPLEX is
  type GETAL is private;
  function "+"            (X,Y : in GETAL)                          return GETAL;
  function "-"            (X,Y : in GETAL)                          return GETAL;
  function "*"            (X,Y : in GETAL)                          return GETAL;
  function "/"            (X,Y : in GETAL)                          return GETAL;
  function MAAK_COMPLEX (REEEL_DEEL       : in FLOAT;
                         IMAGINAIR_DEEL : in FLOAT) return GETAL;
  function REEEL_DEEL      (X : in GETAL)                           return FLOAT;
  function IMAGINAIR_DEEL  (X : in GETAL)                           return FLOAT;
  function HOEK            (X : in GETAL)                           return FLOAT;
  function NORM            (X : in GETAL)                           return FLOAT;
private
  type GETAL is record
                  REEEL     : FLOAT;
                  IMAGINAIR : FLOAT;
                end record;
end COMPLEX;
```

Hoofdstuk 15

2.
```
type SOORT_COMPUTER is (MICRO,MINI,MIDI,SUPER);
package COMPUTER_SET is new SET_PAKKET(SOORT_COMPUTER);
ACHT_BITS_MACHINES            : COMPUTER_SET.SET;
ZESTIEN_BITS_MACHINES         : COMPUTER_SET.SET;
TWEEENDERTIG_BITS_MACHINES    : COMPUTER_SET.SET;
VIERENZESTIG_BITS_MACHINES    : COMPUTER_SET.SET;
```

3.
```
function IS_ONEVEN(S: in SET) return BOOLEAN is
begin
  return ((GETAL_IN(S) mod 2) = 1);
end IS_ONEVEN;
```

Hoofdstuk 16

1. Een pakket dat een taak bevat is een handelend object; een *actor* object. Een pakket zonder taak kan een abstract datatype voorstellen of een automaat, maar beide zijn in beginsel statisch. Als er een taak in een dergelijk pakket wordt opgenomen, dan wordt het dynamisch.

2.
```
task AFZENDER;
task POSTBODE is
  entry NEEM_PAKJE_AAN(P : in TYPE_PAKJE);
end POSTBODE;
```

```
task ONTVANGER is
  entry NEEM_PAKJE_IN_ONTVANGST(P : in TYPE_PAKJE);
end ONTVANGER;

task body AFZENDER is
  PAKJE : TYPE_PAKJE;
begin
  -- vul het pakje
  POSTBODE.NEEM_PAKJE_AAN(PAKJE);
  -- doe nog wat anders
end AFZENDER;
task body POSTBODE is
begin
  loop
    accept NEEM_PAKJE_AAN(P : TYPE_PAKJE) do
    ONTVANGER.NEEM_PAKJE_IN_ONTVANGST(P);
    end NEEM_PAKJE_AAN;
  end loop;
end POSTBODE;
task body ONTVANGER is
  PAKJE : TYPE_PAKJE;
begin
  accept NEEM_PAKJE_IN_ONTVANGST(P : in TYPE_PAKJE) do
    PAKJE := P;
  end NEEM_PAKJE_IN_ONTVANGST;
  -- doe iets anders
end ONTVANGER;
```

Hoofdstuk 17

1.
```
package CONTROLEER_LIJN is
  procedure MONITOR;
  STROOMSTORING : exception;
end CONTROLEER_LIJN;
with MEET_SPANNING;
package body CONTROLEER_LIJN is
  procedure MONITOR is
    FOUTEN : INTEGER := 0;
  begin
    loop
      begin
        MEET_SPANNING;
        exception
          when TE_HOOG_VOLTAGE|TE_LAAG_VOLTAGE =>
            FOUTEN := FOUTEN + 1;
            if FOUTEN = 10 then
              raise STROOMSTORING;
            end if;
      end;
    end loop;
  end MONITOR;
end CONTROLEER_LIJN;
```

4.
```
BIT        : constant := 1;
WOORDEN : constant := 16 * BIT;
type BINARY        is (NUL,EEN);
for  BINARY'SIZE   use 1 * BIT;
for  BINARY        use NUL => 2#0#,
                       EEN => 2#1#);
type WOORDEN       is array (INTEGER range 1 .. 32) of BINARY;
for  WOORDEN'SIZE  use 2 * WOORDEN;
```

Hoofdstuk 18

5. Om de gevraagde wijziging te kunnen aanbrengen voeren we een geheeltallige variabele TIJD_INTERVAL in. Binnen de taak TIJDKLOK schrijven we:

```
if TIKKEN = TIJD_INTERVAL * MINUTEN then
```

Nu kan de waarde van TIJD_INTERVAL naar believen worden aangepast.

Hoofdstuk 20

2. Als controle op correct typegebruik (type checking) niet tussen afzonderlijk gecompileerde eenheden onderling werd uitgevoerd, zou men nooit zeker kunnen zijn, dat eenmaal gekozen abstracties ook verder dwingend door de taal worden opgelegd. De faciliteit van afzonderlijke compileerbaarheid van eenheden zou dan nogal wat aan waarde inboeten. Gelukkig maar, dat Ada wel degelijk een algehele controle op correct typegebruik uitvoert!

Hoofdstuk 21

3. Dit kan het eenvoudigst worden opgelost door een teller toe te voegen in de lus van het hoofdprogramma en deze telkens op te hogen. Voor de select instructie in WAPEN_STATUS moet dan worden toegevoegd:

```
if CYCLUS_TELLER = 3 then
```

Als het resultaat TRUE is voeren we de voorwaardelijke entry-aanroep uit en zetten de teller weer op 0.

4. Dit kan ook op een eenvoudige manier: herhaal de voorwaardelijke entry-aanroep naar TARGET binnen de lus van het hoofdprogramma.

WOORDENLIJST

De volgende woordenlijst is met toestemming van het Ada Joint Program Office (OUSDRE) van het Departement van Defensie van de Verenigde Staten, overgenomen uit: *Reference Manual for the Ada® Programma Language* (Juli 1980), Appendix D. Eraan toegevoegde items zijn met een * gemerkt.

***Abstraction; abstractie**: de wijze waarop wij een grootheid in een probleemgebied beschouwen. Er kan een hiërarchische structuur van abstractieniveaus worden opgebouwd, waarbij een hoger niveau met elementen uit een lager niveau wordt geconstrueerd.

Access type; toegangstype: een type, waarvan de objecten met behulp van een *allocator* of toewijzer worden gecreëerd. Een *access waarde* verwijst naar een dergelijk object.

Aggregate; geaggregeerde grootheid: een verwijzing naar een samengestelde waarde. Een *array aggregate* verwijst naar een waarde van een array type; een *record aggregate* naar een waarde van een record. De samenstellende delen of componenten van een geaggregeerde grootheid kunnen zowel door positionele notatie als door benoeming worden gespecificeerd.

Allocator; toewijzer: creëert een nieuw object van het *access type* en levert een *access waarde* die verwijst naar het gecreëerde object.

***Array; rij**: een samengesteld type met componenten van hetzelfde type en met een indexering via een discrete waarde.

Attribute; kenmerk: een voorgedefinieerd kenmerk of een benoemde grootheid.

***Block; blok**: omvat een (eventueel benoemde) rij instructies, desgewenst voorafgegaan door een declaratiegedeelte.

Body; romp: een programma-eenheid die een subprogramma, pakket of taak implementeert. Een *body stub* is een vervangingselement voor een afzonderlijk gecompileerde body.

***Character; teken**: alle (ASCII) symbolen, waarmee programmatekst kan worden geformuleerd, of resultaten kunnen worden weergegeven.

Collection; collectie: de verzameling toegewezen typen van een *access type*.

***Compatible; verenigbaar**: een declaratie is compatible als de randvoorwaarden overeenkomen met (binnen de grenzen vallen van) die van het basistype.

Compilation unit; compilatie-eenheid: een programma-eenheid die als afzonderlijke tekst voor compilatie wordt aangeboden. De eenheid wordt voorafgegaan door een *contextspecificatie*, waarin de andere compilatie-eenheden worden benoemd, waarvan deze eenheid afhankelijk is.

Component: verwijst naar een gedeelte van een samengesteld object. Een *geïndexeerde component* is een naam die expressies bevat voor indices en benoemt een component van een array of een entry in een familie van entries. Een *geselecteerde component* is de naam van een grootheid gevolgd door een punt en de naam van de component.

Composite type; samengesteld type: een object van een samengesteld type bevat meerdere componenten. Een *array type* is een samengesteld type met componenten, alle van hetzelfde type of subtype; deze componenten kunnen via *indices* worden geselecteerd. Een *record type* is een samengesteld type, waarvan de componenten van verschillende typen kunnen zijn; deze componenten worden via hun namen geselecteerd.

Constraint; randvoorwaarde: een bepreking op de waardenverzameling van een type. Een *range constraint* (intervalbeperking) specificeert onder- en bovengrenzen van een scalair type. Een *accuracy constraint* (precisiebeperking) specificeert de relatief of absoluut toegelaten fout voor waarden van het type real (reëel). Een *index constraint* specificeert onder- en bovengrenzen voor indices. Een *discriminant constraint* specificeert de waarden van de discriminanten van een record of van een grootheid van het type private.

Context specifications; contextspecificaties: gaat vooraf aan een compilatie-eenheid en definieert de compilatie-eenheden, waarvan deze eenheid afhankelijk is en die door de eenheid worden geïmporteerd.

***Conversion; conversie**: de vertaling van een type naar een ander type. In Ada moeten waarden van het ene type expliciet naar die van een ander type worden geconverteerd.

***Declaration; declaratie**: verbindt een naam (identifier) met een gedeclareerde grootheid, zoals een object, een type, een subprogramma, een taak, herbenoemde grootheden, subtypen, pakketten, excepties en generieke eenheden.

***Declarative part; declaratiegedeelte**: een rij declaraties en daarmee samenhangende informatie, zoals subprogrammabodies en representatiespecificaties, die van toepassing zijn in een bepaald gedeelte van een programmatekst. Declaraties worden uitgewerkt in de volgorde waarin zij in de tekst voorkomen. Het bereik (scope) van een grootheid begint in de tekst op de plaats waar deze voor het eerst wordt genoemd. Een grootheid kan daarom alleen eerder gedeclareerde grootheden 'zien'.

Derived type; afgeleid type: een type waarvan de bijbehorende operaties en waarden zijn gedefinieerd door ze over te nemen van een al bestaand type. Dit bestaande type wordt het *ouder* type (parent type) genoemd.

***Designate; verwijzen naar**: een access waarde verwijst bijvoorbeeld naar een toegewezen (allocated) object.

***Designator; verwijzende grootheid**: de naam van een functie. Dit kan ook een operator symbool zijn.

Disambiguation; wegnemen van dubbelzinnigheid: het selecteren van een benoemde grootheid uit een aantal overladen grootheden.

Discrete type; discreet type: een type met een geordende verzameling van verschillende waarden. Het kan hier gaan om het enumeratietype of om het type integer. Discrete typen kunnen gebruikt worden voor indexering en iteratie en voor keuzemogelijkheden in case instructies en record varianten.

Discriminant: een syntactisch te onderscheiden component van een record. De aan- of afwezigheid van een bepaalde record component kan afhangen van de waarde van een discriminant.

Elaboration; uitwerking: de verwerking van een declaratie tijdens het draaien van het programma. Op deze wijze wordt bijvoorbeeld een naam aan een programma-eenheid verbonden of wordt een juist gedeclareerde variabele geïnitialiseerd.

Entity; entiteit of grootheid: wat benoemd kan worden, of waarnaar verwezen kan worden in een programma. Objecten, typen, waarden en programma-eenheden zijn alle entiteiten.

Entry; ingang: gebruikt voor de communicatie tussen taken. Van buitenaf wordt een entry aangeroepen, zoals een subprogramma. Het effect binnen de taak hangt af van de aanwezigheid van één of meer accept-instructies, waarin de acties worden gespecificeerd die moeten worden uitgevoerd als de entry wordt aangeroepen.

Enumeration type; enumeratietype: een discreet type, waarvan de waarden door opsomming worden vastgelegd in de typedeclaratie. De waarden kunnen namen zijn of rijen van tekens.

Error; fout: een situatie waarin gezondigd is tegen een syntactische of semantische regel, of waarin sprake is van een logische tegenstrijdigheid. Er zijn drie categorieën fouten: fouten die tijdens de compilatie ontdekt kunnen worden (fouten tegen taalregels), fouten die tijdens de verwerking kunnen worden ontdekt (excepties) en fouten tegen taalregels waaraan een Ada programma moet voldoen maar die niet door de compiler behoeven te worden gecontroleerd. Een dergelijk programma wordt *onjuist* (erroneous) genoemd en zijn resultaat is onvoorspelbaar.

Exception; exceptie: een gebeurtenis die maakt dat de normale verwerking van het programma wordt onderbroken. Het signaleren van een exceptie wordt in de Ada terminologie *raising an exception* genoemd. Een *exception handler* is een stuk programmatekst dat aangeeft welke actie er naar aanleiding van een exceptie moet worden ondernomen. De exceptie wordt 'afgehandeld'.

Expression; expressie of uitdrukking: een programmagedeelte dat een waarde berekent.

***Generic program unit; generieke programma-eenheid**: een subprogramma of pakket dat werd gespecificeerd met een generiek gedeelte. Een *generieke clausule* bevat de declaratie van generieke parameters. Dit kunnen typen zijn, subprogramma's of objecten. Dit zijn de generieke formele parameters. Bij het creëren van een verschijningsvorm (instantiation) van de eenheid, worden actuele parameters aan de generieke parameters toegevoegd. Een generieke programma-eenheid is een soort blauwdruk voor een klasse van programma-eenheden, waarvan vervolgens verschijningsvormen kunnen worden gecreëerd.

***Hrair limit; Hrair limiet**: het aantal verschillende zaken dat een mens tegelijkertijd kan overzien. Boven deze limiet is het geheel te complex om in één keer te kunnen begrijpen. De term komt uit het boek 'Watership down' (Waterschapsheuvel) van Richard Adams. Voor konijnen is de Hrair limiet vier, voor mensen ongeveer zeven.

***Identifier; naam**: een van de basis bouwstenen van de taal. Een identifier wordt gebruikt voor het benoemen van grootheden en ook voor het benoemen van gereserveerde woorden uit de taal.

Lexical unit; taalelement: het kleinste element in de taal dat betekenis heeft. Bijvoorbeeld: een identifier, een getal, een letterteken of een commentaar.

***Library unit; bibliotheekeenheid**: een compilatie-eenheid, die geen subeenheid van een andere eenheid is. Bibliotheekeenheden zijn onderdeel van een programmabibliotheek.

Literal; letterlijk weergegeven waarde: bijvoorbeeld een letterlijk weergegeven getal, een enumeratiewaarde, een letterteken of een string.

Model number; voorstelbaar getal: een exact voorstelbare waarde van een reëel numeriek type. Operaties op een dergelijk type worden gedefinieerd in termen van de voorstelbare getallen van dit type. De eigenschappen van de voorstelbare getallen en de operaties daarop zijn de minimale eigenschappen die binnen alle implementaties van het reële type behouden blijven.

***Number; getal**: een letterlijke waarde van het type universal_integer of universal_real. Een benoemd getal (named number) is een constant getal waarnaar verwezen mag worden door de identifier van een getalsdeclaratie.

Object: een variabele of een constante. Een object kan naar ieder soort van gegevenselement verwijzen: naar scalaire waarden, naar samengestelde waarden of naar een waarde in een access type.

***Operator**: alle speciale symbolen (zoals "*" of "mod"), die een logische of rekenkundige bewerking uitvoeren op objecten en letterlijk weergegeven waarden.

Overloading; overladen: het hebben van meer dan één betekenis van literals, aggregaten, identifiers en operatoren binnen hetzelfde programmabereik. Een overladen enumeratiewaarde, bijvoorbeeld, is een waarde die in twee of meer enumeratietypen voorkomt. Een overladen subprogramma is een subprogramma, waarvan de naam kan verwijzen naar meer dan één subprogramma, afhankelijk van de gekozen parameters en de resulterende waarden.

Package; pakket: een programma-eenheid die een aantal functioneel samenhangende grootheden bevat. Het *zichtbare* deel van het pakket bevat die grootheden, die van buiten het pakket gebruikt mogen worden. Het *private* deel van het pakket bevat details die voor de gebruiker van het pakket niet van belang zijn, maar die nodig zijn ter aanvulling van de specificatie van de zichtbare grootheden. De pakket*body* bevat de uitwerking (implementatie) van de subprogramma's of taken (mogelijk zelf weer pakketten), zoals in het zichtbare deel gespecificeerd.

Parameter: één van de benoemde grootheden behorend bij een subprogramma, een entry of een generieke programma-eenheid. Een *formele parameter* wordt gebruikt bij de definitie van de body van de eenheid. Een *actuele parameter* is de invulling bij de aanroep van de eenheid. De *parameter modus* geeft aan of het om een invoerparameter, een uitvoerparameter of een invoer/uitvoerparameter gaat. Een *positionele parameter* is een actuele parameter die geïdentificeerd wordt door zijn positie. Een *benoemde parameter* is een actuele parameter die geïdentificeerd wordt door het benoemen van de formele parameter.

Pragma: compiler instructie. Kan gedefinieerd zijn in de taal of via de implementatie.

Private type; afgezonderd type: een type waarvan de structuur en de waardenverzameling exact is gedefinieerd, maar niet bekend kan zijn aan de gebruiker. Alleen de discriminanten en de mogelijke operaties zijn aan de gebruiker bekend. Een type 'private' en de toegelaten operaties worden in het zichtbare deel van een pakket gedefinieerd. Waardetoekenning (assignment) en testen op gelijkheid en ongelijkheid zijn toegelaten, tenzij sprake is van *limited private*.

***Program library; programmabibliotheek**: deel van de APSE programmeeromgeving en herkend door de Ada compiler. Bestaat uit een verzameling compilatie-eenheden.

***Program unit; programma-eenheid**: één van de drie elementaire structuren, waaruit een systeem in Ada is opgebouwd: subprogramma's, pakketten en taken.

Qualified expression; gekwalificeerde expressie: een expressie waaraan een kwalificatie is toegekend via de naam van een type of een subtype. Kan worden gebruikt om het type of subtype van de expressie aan te geven, bijvoorbeeld in het geval van 'overloading'.

Range; bereik: een aaneensluitende rij waarden van een scalair type. Het bereik wordt gespecificeerd door het aangeven van de ondergrens en de bovengrens van de waarden. Kan worden gebruikt om te testen of een waarde tot een bepaald bereik behoort ('in' of 'not in').

***Record**: een samengesteld type dat bestaat uit een aantal componenten, die van verschillend type kunnen zijn. Naar een component kan worden verwezen door middel van de *puntnotatie* (selected component notation).

Rendez-vous; rendez-vous of ontmoeting: de wisselwerking tussen twee parallelle taken, die begint als de ene taak de entry van de andere aanroept en de andere taak een accept-instructie ten gunste van de aanroepende taak uitvoert.

Representation specification; specificatie van voorstellingswijze: specificeert de afbeelding van een datatype naar een ander datatype en is machine-afhankelijk. Het is mogelijk de afbeeldingswijze volledig te specificeren, of slechts een criterium voor de keuze van een afbeeldingswijze aan te geven.

Scalar types; scalaire typen: een type met niet samengestelde waarde. Omvat discrete typen (enumeratietypen en integer typen) en reële typen.

Scope; reikwijdte: dat deel van een programmatekst, waarin een declaratie geldig is heet de 'scope' van die declaratie.

***Semantics; semantiek:** de betekenis van een bepaalde structuur of grootheid.

***Statement; instructie:** in tegenstelling tot een declaratie, die een bepaalde grootheid definieert, leidt de verwerking van een instructie tot het uitvoeren van een bepaalde actie.

Static expression; statische expressie; een expressie, waarvan de waarde niet afhangt van tijdens de verwerking van het programma berekende waarden.

Subprogram; subprogramma; een voor verwerking geschikt programma-eenheid, mogelijk met één of meer parameters voor het doorgeven van waarden, vanuit de plaats waar het subprogramma wordt aangeroepen. Een *subprogrammadeclaratie* specificeert de naam van het subprogramma en de parameters. De *subprogrammabody* specificeert de wijze van verwerking. Een subprogramma kan een *procedure* zijn of een *functie*. Een procedure voert een actie uit; een functie levert een waarde op.

Subtype: wordt uit een type verkregen door een beperking op te leggen aan de waardenverzameling behorend bij dat type. De toegelaten operaties op het subtype zijn dezelfde als de operaties behorend bij het basistype.

***Subunit; subeenheid:** de body van een subprogramma, een pakket of een taak, die binnen een andere compilatie-eenheid gedeclareerd werd.

Syntax; syntaxis: de regels die aangeven hoe in een taal toegelaten uitdrukkingen eruit mogen zien. Appendix A geeft met behulp van *syntaxdiagrammen* constructieregels voor de toegelaten uitdrukkingen in Ada.

Task; taak: een programma-eenheid, die tegelijkertijd (parallel) met andere programma-eenheden actief kan zijn. De *taakspecificatie* geeft de naam en de parameters van een taak; de *taakbody* specificeert de werking. Via een *taaktype* kan een aantal taken van hetzelfde type worden gedeclareerd. De verwerking van een bepaalde programma-eenheid wordt pas beëindigd als alle taken, die erin gedeclareerd werden, beëindigd zijn. Een taak heet *voltooid* ('completed') als de taak aan het eind van zijn body wacht op een andere taak, of als de taak *afgebroken* is ('aborted'), maar nog niet *beëindigd* ('terminated') Een voltooide taak kan niet meer worden aangeroepen; een beëindigde taak is niet meer actief.

Type: bepaalt een waardenverzameling en een verzameling op die waarden toegelaten bewerkingen. Een *typedefinitie* introduceert een nieuw uniek type; een subtype creëert een (mogelijkerwijs) beperkte variant van een basistype. Een *typedeclaratie* verbindt een naam met een type, dat via een typedefinitie werd ingevoerd.

Use clause; use clausule: maakt de declaraties in het zichtbare deel van een pakket vanuit een andere eenheid toegankelijk.

Variant: deel van een record, dat afhankelijk van de record discriminant verschillende recordcomponenten kan specificeren. Elke mogelijke waarde van de discriminant bepaalt precies één waarde van de variant.

Visibility; zichtbaarheid: de declaratie van een grootheid met een bepaalde naam is vanuit een bepaalde plaats in de programmatekst *zichtbaar* als aan deze naam op die plaats een betekenis kan worden toegekend.

NOTEN

PAKKET 1: Probleemstelling

[1] E.W. Dijkstra, "The Humble Programmer" (Turing Award Lecture), *Communications of the ACM*, Vol. 15, No. 10 (October 1972): 861. Copyright 1972, Association for Computing Machinery, Inc., reprinted by permission.

HOOFDSTUK 1: Inleiding

[1] D.A. Fisher, "A Common Programming Language for the Department of Defense - Background and Technical Requirements", Institute for Defense Analysis, Report P-1191 (June 1976): 19.

[2] I.C. Pyle, *The Ada Programming Language* (Englewood Cliffs N.J.: Prentice-Hall, Inc., 1981), p. ix. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, N.J.

[3] D.J. Foss and D.T. Hakes, *Psycholinguistics; An Introduction to the Psychology of Language* (Englewood Cliffs, N.J.: Prentice-Hall, 1978), p. 385. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, N.J.

[4] Fisher, *Background and Technical Requirements*, p. 8.

HOOFDSTUK 2: De Softwarecrisis

[1] *Zie ook* C.A.R. Hoare, "Professionalism" (Invited talk given at BCS-81, British Computing Society, July 1, 1981).

[2] W.A. Wulf, "Languages en Structured Programs", in Raymond Yeh, ed., *Current Trends in Computer Programming*, Vol. 1 (Englewood Cliffs, N.J.: Prentice-Hall, Inc.), p. 33. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, N.J., and Wm.A. Wulf, Tartan Laboraties, Pittsburgh, PA.

[3] Dijkstra, *The Humble Programmer*, p. 863.

[4] Fisher, *Background and Technical Requirements*, p. 2.

[5] Ibid., pp. 2-3.

[6] B.W. Boehm, "Software and Its Impact: A Quantitative Assessment", *Datamation*, May 1973, p. 13. Reprinted with permission of DATAMATION® magazine, © Copyright by Technical Publishing Company, A Dun & Bradstreet Company, 1983. All rights reserved.

[7] M.T. Devlin, Introducing Ada: Problems and Potentials, USAF Satellite Control Facility (unpublisged report), 1980, p. 2.

[8] Ibid., p. 2.

[9] Wulf, op. cit., p. 34.

[10] *Zie ook* E.W. Dijkstra, "Programming Considered as a Human Activity", (Paper presented to the International Federation of Information Processing Conference, New York, September 1965), p. 6.

[11] Brooks, THE MYTHICAL MAN MONTH, © 1975, Addison-Wesley, Reading, MA, pp. 94. Reprinted with permission.

HOOFDSTUK 3: Ada's Ontwikkelingsgeschiedenis

[1] Wulf, op. cit. p. 60.

[2] Fisher, *Background and Technical Requirements*, p. 5.

[3] Ibid., p. 3.

[4] W.A. Whitaker, private communication.

[5] Fisher, *Background and Technical Requirements*, p. 6.

[6] W.A. Whitaker, The U.S. Department of Defense Common High Order Language Effort, SIGPLAN Notices, February 1978, p. 2.

[7] Fisher, *Background and Technical Requirements*, pp. 21-23.

[8] Ibid., p. 4.

[9] Whitaker, *High Order Language Effort*, p. 4.

[10] Requirements for High Order Programming Languages, STRAWMAN, Department of Defense, August 1975.

[11] Requirements for High Order Programming Languages, WOODENMAN, Department of Defense, August 1975.

[12] Requirements for High Order Programming Languages, TINMAN, Department of Defense, June 1976.

[13] Fisher, *Background and Technical Requirements*, p. 10.

[14] DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems, October 26, 1976.

[15] DoD Directive 5000.31, Interim List of DoD Approved High Order Languages (HOL), November 24, 1976.

[16] *See also*: S. Amoroso, P. Wegner, D. Morris, and D. White, Lamguage Evaluation Coordinating Committee Report to the High Order Language Working Group, January 14, 1977.

[17] Whitaker, *High Order Language Effort*, pp. 7-8.

[18] Requirements for High Order Programming Languages, IRONMAN, Department of Defense, January 14, 1977.

[19] W.E. Carlson, L.E. Druffel, D.A. Fisher and W.A. Whitaker, "Introducing Ada" (Paper presented at ACM-80, October 1980), p. 264. Copyright 1972, Association for Computing Machinery. Inc., reprinted by permission.

[20] W.E. Carlson, "Ada: A Standard Language for Defense Systems", *Signal* (March 1980): 25. Reprinted by permission from SIGNAL, the official journal of the Armed Forces Communications and Electronics Association, Copyright 1980.

[21] Requirements for High Order Programming Languages, Revised IRONMAN, Department of Defense, July 1977.

[22] J.N. Buxton, L.E. Druffel, and T.A. Standish, "Recollections on the History of Ada Environments", *Ada Letters*, Vol. 1, No. 1 (July/August 1981): I-1.16. With permission from Ada Letters, a publication of Ada Tech.

[23] Requirements for the Programming Environment for the Common High Order Language, SANDMAN, Department of Defense, July, 1978.

[24] Requirements for the Programming Environment for the Common High Order Language, PEBBLEMAN, Department of Defense, July 1978.

[25] Requirements for High Order Programming Languages, STEELMAN, Department of Defense, June 1978.

[26] *See also* D.L. Moore, *Ada, Countess of Lovelace, Byron's Legitimate Daughter*, New York: Harper & Row, 1977.

[27] Carlson, Druffel, Fisher, and Whitaker, "Introducing Ada", p. 265.

[28] Ibid., p. 256.

[29] Requirements for the Programming Environment for the Common High Order Language, STONEMAN, Department of Defense, February 1980. *See also* J.N. Buxton and L.E. Druffel, "Requirements of an Ada Programming Environment: Rationale for STONEMAN", (Proceedings of IEEE Computer Society's International Computer Software and Applications Conference, Chicago, Ill., October 29, 1980), pp. 66-72.

[30] P.M. Cohen, "From HOLWG to AJPO - Ada in Transition", *Ada Letters*, Vol 1, No. 1 (July/August 1981): I-1.23. With permission from Ada Letters, a publication of Ada Tech.

[31] R. Abbott, Ada Style Guide, Ada Joint Program Office, November 1981.

[32] Strategy for Ada Education and Training, Ada Joint Program Office, November 1981.

PAKKET 2: Hier Is Ada

1 Genesis 11:6, King James Version. *Zie ook* Genesis 4:19-23 and Genesis 36:2-16.

HOOFDSTUK 4: Software Ontwikkelingsmethoden

[1] D.T. Ross, J.B. Goodenough, and C.A. Irvine, "Software Engineering: Process, Principles, and Goals", *Computer* (May 1975): 65.

[2] Ibid.

[3] Ibid.

[4] Ibid., p. 66

[5] Ibid.

[6] Ibid.

[7] Devlin, Introducing Ada, p. 5.

[8] Ross, Goodenough, and Irvine, "Software Engineering", p. 67.

[9] Ibid.

[10] Ibid.

[11] Ibid., p. 66

[12] Ibid., p.67.

[13] E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Programs and System Design* (Englewood Cliffs, N.J.: Prentice-Hall, 1979), p. 85. Reprinted with permission of Prentice-Hall, Inc., Englewood Cliffs, N.J.

[14] Ibid., p. 106.

[15] Ross, Goodenough, and Irvine, "Software Engineering", p. 67.

[16] Ibid.

[17] Ibid.

[18] *Zie ook* G.A. Miller, "The Magical Number Seven, Plus or Minus Two", *The Psychological Review*, Vol. 53, No. 2 (March 1956).

[19] Yourdon, *Structured Design*, p. 69.

[20] *Zie* R. Adams, *Watership Down* (New York: Macmillan, 1972), p. 13.

[21] Yourdon, *Structured Design*, p. 106.

[22] Ibid., p. 246.

[23] *See also* M. Jackson, *Principles of Program Design*, New York: Academic Press, 1975.

[24] *Zie ook* D. Parnas, "On the Criteria to be Used in Decomposing a System into Modules", CMU-CS-71-101, *Communications of the ACM*, Vol. 15, No. 12 (December 1972).

[25] *Zie ook* D.T. Ross and J.R. Schoman, "Structured Analysis for Requirements Definition", *IEEE Transactions on Software Engineering*, January 1977.

[26] *Zie ook* C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Los Angeles: Improved Systems Technologies, July 1977.

[27] *Zie ook* W.P. Stevens, G.J. Myers, and L. Constantine, "Structured Design", *IBM Systems Journal*, Vol. 13, No. 2, February 1974.

[28] *Zie ook* D.D. McCracken, "Revolution in Programming: An Overview", *Datamation*, December 1973.

[29] P. Wegner, The Ada Programming Language and Environment, unpublished draft, June 6, 1981.

HOOFDSTUK 5: Objectgericht Ontwerp

[1] H. Ledgard and M. Marcotty, *The Programming Language Landscape* (Chicago: Science Research Associates, 1981), p. 166.

[2] R. Balzer, N. Goldman, and D. Wile, "Informality in Program Specifications", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2 (March 1978), p. 94.

[3] *Zie ook* Ada Style Guide, pp. 137-146 and Abbott, R.J., *Program Design by Informal English Description*, an unpublished report, 1982 (to appear in *Communications of the ACM* in 1983).

HOOFDSTUK 6: Ada, Een Overzicht

[1] *Reference Manual for the Ada Programming Language*, United States Department of Defense, Ada Joint Program Office, July 1982.

[2] G. Booch, "Object-Oriented Design", *Ada Letters*, Vol. 1, No. 3 (March/April 1982): 64.

[3] G. Booch, "Describing Software Design with Ada", *SIGPLAN Notices*, September 1981.

PAKKET 3: Datastructuren

[1] With permission from C.A.R. Hoare, "Notes on Data Structuring", in O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* (London: Academic Press, 1972), p. 83. Copyright: Academic Press Inc. (London) Ltd.

HOOFDSTUK 7: Eerste Ontwerpprobleem: Tellen Van Blaadjes

[1] G. Booch, "Object-Oriented Design", *Ada Letters*, Vol. 1, No. 3:56. *See also* R.J. Abbott, "Report on Teachin Ada", Technical Report SAI-81-313-WA, Science Applications, Inc., McLean, Virginia, December 1980.

HOOFDSTUK 8: Abstract Voorstellen Van Gegevens En Ada's Datatypen

[1] J. Ichbiah, J. Barnes, and R. Firth, *Ada Programming Course* (La Cella Saint Cloud, France: ALSYS, 1980), p. 38.

PAKKET 4: Algoritmen En Besturing

[1] W. Shakespeare, *Hamlet*, act 3, sc. 2, line 17.

HOOFDSTUK 11: Expressies En Instructies

[1] D.A. Fisher, "A Common Programming Language for the Department of Defense - Background and Technical Requirements", Institute for Defense Analysis, Report P-1191 (June 1976): 19.

[2] C. Boehm and G. Jocopini, "Flow Diagrams, Turing Machines and Laguages With Only Two Formation Rules", *Communications of the ACM*, May 1966, pp. 366-371. Copyright 1966, Association for Computing Machinery, Inc., reprinted by permission.

PAKKET 5: De Pakket-Aanpak

[1] Uit *Godel, Escher, Bach: An Eternal Golden Braid* by Douglas R. Hofstadter. Copyright © 1979 by Basic Books, Inc. By permission of Basic Books, Inc., Publishers, New York.

PAKKET 6: Parallelle Real-Time Verwerking

[1] W. Shakespeare, *Macbeth*, act 1, sc. 7, line 1.

HOOFDSTUK 16: Taken

[1] C.A.R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8 (August 1978): 666.

PAKKET 7: Systeemontwikkeling

[1] Copyright © by John Gall. Reprinted by permission of TIMES BOOKS, a division of Quadrangle/The New York Times Book Co., Inc., from *Systemantics: How Systems Work and Especially How They Fail* by John Gall.

HOOFDSTUK 20: Programmeren In Het Groot

[1] J. Ichbiah, J. Barnes, and R. Firth, *Ada Programming Course* (La Cella Saint Cloud, France: ALSYS, 1980), p. 212.

PAKKET 8: Programmeren Met Ada

[1] Uit *The Psychology of Computer Programming* by G.M. Weinberg. Copyright © 1971 by Van Nostrand Reinhold Company. Reprinted by permission of the publisher, p. 209.

HOOFDSTUK 22: De Ada Programmeeromgeving

[1] Requirements for Ada Programming Support Environments, STONEMAN, Department of Defense, February 1980, p. 1.

[2] *Zie ook* J.N. Buxton and L.E. Druffel, "Requirements for an Ada Programming Environment: Rationale for STONEMAN", (Proceedings of IEEE Computer Society's International Computer Software and Applications Conference, Chicago, Ill., October 29-31, 1980), pp. 66-72.

[3] Notes on Ada Programming Support Environments, SofTech, August 11, 1981, p. 418/4.

[4] M. Wolf, W. Babich, R. Simpson, R. Tholl, and L. Weissman, "The Ada Language System", *Computer* (June 1981): 38 © IEEE 1981.

HOOFDSTUK 23: Ada En De Software Levenscyclus

[1] T.J. Wheeler, "Embedded Systems Design with Ada as the System Design Language", CORADCOM, 1982.

[2] PDL/ADA: DSM Project's Ada - Program Design Language Reference Manual, IBM Federal Systems Division, April 22, 1981.

[3] H. Hart, "Ada for Design: An Approach for Transitioning Industry Software Developers", *Ada Letters*, Vol. 2, No. 1 (July/August 1982): 50.

HOOFDSTUK 24: Toekomstige Ontwikkelingen En Conclusies

[1] D.G. Stephen, D. Doris, B. Barbazette, L. Johnson, and B. Murthpy, "DoD Digital Data processing Study: A Ten year Forecast", Electronics Industries Association, 1980.

[2] P.W. Wegner, The Ada Language and Environment, draft, June 1981, p. 13.

[3] Whitaker, High Order Language Effort, p. 11.

[4] D.A. Fisher, "A Common Programming Language for the Department of Defense - Background and Technical Requirements", Institute of Defense Analysis, Report P-1191 (June 1976): 6.

BIBLIOGRAFIE

ALGEMEEN

A Common Language for Computers, *Business Week*, March 23, 1981, pp. 84B-84E.

Ada Course Notes, Georgia Institute of Technology, Atlanta, 1980

Ada - A Report to the Department of Industry, Department of Industry, Washington, D.C., May 1979.

General Trends in the DoD, briefing to Automatic Data Processing Single Managers Conference, USAF Academy, Colorado Springs, Colorado, January 1981.

Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, Boston, December 1980.

Proceedings of the Ada Debut, Defense Advanced Research Projects Agency, Washington, D.C., Report AD-A095 569/0, September 1980.

Barnes, J., An Overview of Ada, *Software Practices and Experience*, Vol. 10, 1980.

Barnes, J., *Programming in Ada*, Addison-Wesley, London, 1981.

Booch, G., and Bolz, D., *Software Engineering with Ada (course notes)*, Department of Computer Science, USAF Academy, Colorado Springs, Colorado, 1981.

Booch, G., *Software Engineering with Ada*, Proceedings, 15th IEEE Conference on Circuits, Systems, and Computers, Asilomar, Calif., November 1981.

Booch, G., *Ada Promotes Software Reliability with Pascal-like Simplicity, EDN*; January 7, 1981.

Booch, G., *Introducing Ada*, Proceedings of American Institute of Aeronautics and Astronautics Conference, Dayton, Ohio, November 1981.

Bowles, Kenneth L., *The Impact of Ada on Software Engineering*. Paper presented at National Computer Conference, Houston, June 1982.

Braun, C.L., Ada: Programming in the 80's, *Computer*, June 1981.

Brender, R.F., and Nassi, I.R., What is Ada?, *Computer*, June 1981.

Carlson, W.E., Ada: A Promising Beginning, *Computer*, June 1981.

Carlson, W.E., Ada: A Standard Language for Defense Systems, *Signal*, March 1980.

Carlson, W.E., Druffel, L.E., Fisher, D.A., and Whitaker, W.A., *Introducing Ada*. Proceedings of ACM-80, October 1980.

Cohen, P.M., From HOLWG to AJPO - Ada in Transition, *Ada Letters*, Vol. 1, No. 1, July/August 1981.

Cornhill, D., and Gordon, M.E., Ada - The Latest Words in Process Control, *Electronic Design*, September 1, 1980.

Devlin, M.T., *Introducing Ada: Problems and Potentials*, USAF Satellite Control Facility, Sunnyvale, Calif., 1980.

Estel, R.G., A Chapter in the History of DoD-1, *SIGPLAN Notices*, March 1978.

Filipski, G.L., Moore, D.R., and Newton, J.E., Ada as a Software Transition Tool, *SIGPLAN Notices*, November 1980.

Fisher, D.A., *The Common Programming Language Effort in the Department of Defense*. Paper presented at Computers in Aerospace Conference, October 31, 1977.

Fisher, D.A., DoD's Common Programming Language Effort, *Computer*, March 1978.

Glass, R.L., From Pascal to Pebbleman ... and Beyond, *Datamation*, July 1979.

Gross, S., Ada Language Finds Wide Acceptance, *Electronic News*, September 22, 1980.

Halloran, R., Pentagon Pins Its Hopes on Ada, *New York Times*, November 30, 1980.

Hibbard, P., Hisgen, A., Rosenberg, J., and Sherman, M., *Programming in Ada: Examples*, Report CMU-CS-80-149, Department of Computers Science, Carnegie-Mellon University, Pittsburgh, Pa., October 1980.

Hibbard, P., Hisgen, A., Rosenberg, J., and Sherman, M., *Programming in Ada: Principles, Concepts, Techniques*, Report CMU-CS-80-149, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1980.

Hibbard, P., Hisgen, A., Rosenberg, J., and Sherman, M., *Studies in Ada Style*, Springer-Verlag, New York, 1981.

Ichbiah, J., Barnes, J., and Firth, R., *Ada Programming Course*, La Cella Saint Cloud, France, December 1980.

Johnson, R.C., Ada, the Ultimate Language?, *Electronics*, February 10, 1981.

Kamijo, F., *A New Programming Language - HOL project of DoD*, Information Processing Society of Japan, Tokyo, 1978.

Kling, R., and Scacchi, W., The DoD Common High Order Programming Language Effort (doD-1): What Will the Impacts Be?, *SIGPLAN Notices*, February 1979.

LeBlanc, R.J., and Goda, J.J., *The Impact of Ada on Software Development*. Proceedings of SOUTHEASTCON 81, April 1981.

Ledgard, H., *Ada - An Introduction*, Springer-Verlag, New York, 1981.

Lomuto, N., *Early Experiences with Ada Tasks*. Paper presented to National Computer Conference, Houston, June 1982.

Loveman, D., Ada: How Big a Difference Will It Make in Software?, *Military Electronics/Countermeasures*, May 1981.

Mathis, R.F., Names for Programming Languages: A Dubious Analysis, *Ada Letters*, March/April 1981.

Mayoh, B., *Problem Solving with Ada*, Wiley, New York, 1981.

Moore, David, *Ada Course Notes*, Integrated Computer Systems, Los Angeles, 1981.

Pyle, I.C., *The Ada Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

Rymer, J., An Ada Tutorial, *IBM Software Engineering Exchange*, October 1980.

Schwartz, Larry, *Ada Course Notes*, IBM Federal Systems Division, Bethseda, Md., 1981.

Waugh, D.W., Ada as a Design Language, *IBM Software Engineering Exchange*, October, 1980.

Wegner, P.W., An Ada Self-Assessment, *Communications of the ACM*, Vol. 24, No. 10, October 1981.

Wegner, P.W., Programming with Ada - An Introduction by Means of Graduated Examples, *SIGPLAN Notices*, December 1979.

Wegner, P.W., *Programming with Ada - An Introduction by Means of Graduated Examples*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

Wegner, P.W., The Ada Language and Environment, *SIGSOFT Software Engineering Notes*, April 1980.

Wegner, P.W., *The Ada Language and Environment*, unpublished report, June 1981.

Werner, F., Is Ada the Programming Language for the '80's?, *Computerworld*, October 5, 1981.

Wheeler, T.J., *Embedded Systems Design with Ada as the System Design Language*, Army CORADCOM, Ft. Monmouth, N.J., 1980.

Whitaker, W.A., *Ada - The DoD Common High Order Language*, National Aerospace and Electronics Conference, 1979.

Whitaker, W.A., *Ada - The New DoD Standard High Order Language*. 1979 Summer Computer Simulation Conference, July 1979.

Whitaker, W.A., The U.S. Department of Defense Common High Order Language Effort, *SIGPLAN Notices*, February 1978.

Wolf, M.I., Babich, W., Simpson, R., Tholl, R., and Weissman, L., The Ada Language System, *Computer*, June 1981. © IEEE 1981.

TAAL SPECIFIEK

Ada Compiler Validation Capability, SofTech, Inc., Waltham, Mass., Report 1067-1.1, February 1980.

Ada Language Reference Card, Intermetrics Inc., Cambridge, Mass., March 1981.

Ada Test and Evaluation, Intermetrics Inc., Cambridge, Mass., Report IR-663, February 1981.

Ada Test and Evaluation Workshop. Defense Advanced Research Projects, Washington, D.C., October 23-26, 1979.

Diane Reference Manual, Institut für Mathematik II, Universität Karlsruhe and Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., Report 1/81, March 1981.

DoD Common High Order Language, Phase I Reports and Analyses, High Order Language Working Group, Washington, D.C., June 1978.

DoD Common High Order Language, Phase II Reports and Analyses, Defense Advanced Research Projects Agency, Washington, D.C., Report ADA-80-1-M, January 1980.

Engineering Specifications of the iAPX 432 Extensions to Ada, Intel Corp., Aloha, Oreg., Manual 171871-001, January 1981.

iAPX 432 Object Primer, Intel Corp., Aloha, Oreg., Manual 171858-001 Rev. B, 1980.

IBM PDL/ADA Language Refernce Card, IBM Federal Systems Division, Bethseda, Md., May 1981.

PDL/ADA:DSM Project's Ada - Program Design Language Reference Manual, IBM Federal Systems Division, Bethseda, Md., April 1981.

PDL/Ada, *IBM Software Engineering Exchange*, Vol. 2, No. 1, October 1980.

Red/Green Evaluation Report, Department of Industry, Washington, D.C., April 1979.

Using Selected Features of Ada: A Collection of Papers, Software Technology Development Division, CENTACS Center for Tactical Computer Systems, US Army, Communications Electronics Command, Ft. Monmouth, N.J., 1981.

Abbott, R., *Report on Teaching Ada*, Science Applications, Inc., Los Angeles, Report SAI-81-312-WA, December 1980.

Arnold, R.D., The Nebula Architecture: Ada Issues, *Ada Letters*, May/June 1981.

Bjorner, D., And Oest, O.N., eds., *Towards a Formal Description of Ada*, Springer-Verlag, New York, 1980.

Boehm, B., and Jocopini, G., Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules, *Communications of the ACM*, May 1966.

Booch, G., Describing Software Design in Ada, *SIGPLAN Notices*, September 1981.

Clapp, J.A., Loebenstein, E., and Rhymer, P., *A Cost/Benefit Analysis of High Order Language Standardization*, Mitre Corp., Washington, D.C., Report P 78-206, September 1977.

Cole, S.N., Ada Syntax Cross Reference, *SIGPLAN Notices*, March 1981.

Dahl, F., Dijkstra, E.W., and Hoare, C.A.R., *Structured Programming*, Academic Press, new York, 1972.

Dijkstra, E.W., DoD-1: The Summing up, *SIGPLAN Notices*, July 1978.

Druffel, L.E., Ada - How Will It Affect College Offerings?, *Interface*, September 1979.

Duncan, A.G., and Hutchinson, J.S., Using Ada for Industrial Embedded Microprocessor Applications, *SIGPLAN Notices*, November 1980.

Galkowski, J.T., A Critique of the DoD Common Language Effort, *SIGPLAN Notices*, Vol. 15, No. 6, June 1980.

Giese, C., and Mitchell, J., *Ada - A Suitable Replacement for COBOL?*, U.S. Army Institute for Research, Ft. Monmouth, N.J., February 1981.

Good, D.I., Young, W.D., and tripathi, A.R., *An Evaluation of the Verifiability of Ada*, unpublished report, September 1980.

Goodenough, J.B., The Ada Compiler Validation Capability, *Computer*, June 1981.

Goos, G., and Hartmanis, J., eds., *Design and Implementation of Programming Languages, DoD Sponsored Workshop, Ithaca 1976*, Springer-Verlag, New York, 1977.

Groves, L.J., and Roger, W.J., The Design of a Virtual Machine for Ada, *SIGPLAN Notices*, November 1980.

Hart, H., *Ada for Design: An Approach for Transitioning Industry Software Developers*. Paper presented at NSIA Software Group Conference, October 14, 1981.

Hoare, C.A.R., Communication Sequential Processes, *Communications of the ACM*, Vol. 21, No. 8, August 1978.

Hoare, C.A.R., The Emperor's New Clothes, *Communications of the ACM*, February 1981.

Ichbiah, J.D., *Ada and the Development of Software Components*. Proceedings of the 4th International Conference on Software Engineering, September 1979.

Knobe, B., Flight Languages: Ada vs HAL/S, *Journal for Guidance and Control*, January/February 1981.

Loveman, D., Ada Defines Reliability as a Basic feature, *Electronic Design*, September 27, 1980.

Loveman, D., Ada Knack for Multitasking Benefits Process Control, *Electronic Design*, December 6, 1980.

Loveman, D., Ada Resolves the Unusual with 'Exceptional' Handling, *Electronic Design*, January 22, 1981.

Loveman, D., Subprograms and Types Boost Ada Versatility, *Electronic Design*, October 25, 1980.

Luckum, D.C., Larsen, H.J., Stevenson, D.R., and von Henke, F.W., *Ada-M: An Ada-based Medium-level Language for Multiprocessing*, Stanford University, December 7, 1980.

Rattner, J., and Lattin, W.W., Ada Determines Architecture of 32-bit Microprocessor, *Electronics*, February 24, 1981.

Scheer, L., and McClimers, M., *DoD's Ada Compared to Present Military Standard HOLs: A Look at New Capabilities*, Systems Consultants, Inc., Dayton, Ohio, 1980.

Whitaker, W.A., *Professor Dijkstra's SIGPLAN Letter*, Defense Advance Research Projects Agency Memorandum for the Record, October 26, 1978.

Winterstein, G., Peusih, G., Drossopoulos, S., and Dausmann, *Ada Documentation and Programming Guidelines*, Institut für Informatik II, Universität Karlsruhe, September 1981.

Zeigler, S., Allegre, N., Johnson, R., Morris, J., and Burns, G., Ada for the Intel 432 Microcomputer, *Computer*, June 1981.

PROGRAMMEEROMGEVING

Ada Environment Workshop, DoD High Order Language Working Group, Washington, D.C., November 27, 1979.

Buxton, J.N., Druffel, L.E., and Standish, T.A., Recollections on the History of Ada Environments, *Ada Letters*, Vol. 1, No. 1, July/August 1981.

Buxton, J.N., and Druffel, L.E., *Requirements for an Ada Programming Support Environment; Rationale for STONEMAN*. Proceedings of COMPSAC, October 1980.

Elzer, P.F., *Some Observations Concerning Existing Software Environments*, DORNIER Systems, unpublished DoD report, May 1979.

Fisher, D.A., *Design Issues for Ada Program Support Environments: A Catalog of Issues*, Science Applications Inc., Washington, D.C., Report SAI-81-289-WA, October 1980.

Loveman, D., The Ada International Environment: An Introduction to the Problem, *Ada Letters*, March/April, 1981.

Standish, T.A., *Proceedings of the Irvine Workshop on Alternatives for the Environment, Certification, and Control of the DoD Common High Order Language*, Department of Information and Computer Science, University of California, Irvine, Calif., Report UCI-ICS-78-83, June 1978.

Standish, T.A., *The Importance of Ada Programming Support Environments*. Paper presented at National Computer Conference, Houston, June 1982.

Stenning, V., Froggatt, T., Gilbert, R., and Thomans, E., The Ada Environment: A Perspective, *Computer*, June 1981.

SPECIFICATIES

Ada Compiler Validation Implementers' Guide, SofTech Inc., Waltham, Mass., Report 1067-2.3, October 1, 1980.

Ada Programming Language, Department of Defense, Washington, D.C., ANSI/MIL-STD 1815A-1983.

DoD Directive 5000.29, *Management of Computer Resources in Major Defense Systems*, October 26, 1976.

DoD Directive 5000.31, *Interim List of DoD Approved High Order Languages (HOL)*, November 24, 1976.

Formal Definition of the Ada Programming Language, Honeywell Inc., Cii, France, Honeywell-Bull and Inria, November 1980.

Informal Language Specification (Red), Intermetrics Inc., Cambridge, Mass., March 1979.

MIL-STD-483 (USAF), *Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, Note 2*, March 21, 1979.

MIL-STD-490 (USAF), *Specifications Practices*, October 30, 1968.

Preliminary Ada Reference manual, *SIGPLAN Notices*, June 1979.

Rationale for the Design of the Green Programming Language, Honeywell and Cii Honeywell-Bull, February 15, 1978.

Red Language Design Rationale, Intermetrics Inc., Cambridge, Mass., Report IR-382, March 1979.

Red Language Refernce Manual, Intermetrics Inc., Cambridge, Mass., Report IR-310.2, March 1979.

Reference Manual for the Ada Programming Language, Ada Joint Program Office, Department of Defense, Washington, D.C., October 1982.

Reference Manual for the Green Programming Language, Honeywell Inc. and Cii Honeywell-Bull, March 15, 1979.

Requirements for High Order Programming Languages, STRAWMAN, Department of Defense, Washington, D.C., July 1975.

Requirements for High Order Programming Languages, WOODENMAN, Department of Defense, Washington, D.C., August 1975.

Requirements for High Order Programming Languages, TINMAN, Department of Defense, Washington, D.C., June 1976.

Requirements for High Order Programming Languages, IRONMAN, Department of Defense, Washington, D.C., January 1977.

Requirements for High Order Programming Languages, Revised IRONMAN, Department of Defense, Washington, D.C., July 1977.

Requirements for High Order Programming Languages, STEELMAN, Department of Defense, Washington, D.C., June 1978.

Requirements for the Programming Environment for the Common High Order Language, PEBBLEMAN, Department of Defense, Washington, D.C., July 1978.

Requirements for the Programming Environment for the Common High Order Language, Revised PEBBLEMAN, Department of Defense, Washington, D.C., January 1979.

Requirements for the Programming Environment for the Common High Order Language, Preliminary STONEMAN, Department of Defense, Washington, D.C., November 1979.

Requirements for the Programming Environment for the Common High Order Language, STONEMAN, Department of Defense, Washington, D.C., February 1980.

Strategy for Ada Education and Training, Ada Joint Program Office, Department of Defense, Washington, D.C., November 1981.

Fisher, D.A., *A Common Programming Language for the Department of Defense - Background and Technical Issues*, Report P-1191, Institute for Defense Analysis, Arlington, Va., June 1976.

PROGRAMMA-ONTWIKKELING

Abbott, R.J., *Program Design by Informal Description*, unpublished report, 1982 (to appear in *Communications of the ACM* in 1983).

Balzer, R., Goldman, N., and Wile, D., Informality in Program Specification, *IEEE Transactions of Software Engineering*, Vol. SE-4, No. 2, March 1978.

Boehm, B.W., Software and Its Impact: A Quantitative Assessment, *Datamation*, May 1973.

Booch, G., *Object Oriented Design*, USAF Academy, Colo., 1980. *See also Ada Letters*, Vol. 1, No. 3, March/April 1982.

Dijkstra, E.W., The Humble Programmer, *Communications of the ACM*, Vol. 12, No. 10, October 1972.

Gare, C., and Sarson, T., Structured Systems Analysis: Tools and Techniques, *Computer*, July 1977.

Hart, H., *Ada for Design: An Approach for Transitioning Industry Software Developers*, NSIA Software Group Conference, Pctober 14, 1981.

Jackson, M., The Jackson Design Methodology, Infotec State of the Art Report, Structured Programming, 1978.

Jensen, R.W., Structured Programs, *Computer*, March 1981.

McCracken, D.D., Revolution in Programming, *Datamation*, December 1973.

Parnas, D.L., *A Paradigm for Software Module Specification with Examples*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., March 1971.

Parnas, D.L., *Information Distribution Aspects of Design Methodology*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., February 1971.

Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules*, Report CMU-CS-71-101, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August 1971.

Ross, D.T., and Brackett, J.W., An Approach to Structured Analysis, *Computer Decisions*, September 1976.

Ross, D.T., Goodenough, J.B., and Irvine, C.A., Software Engineering: Process, Principles, and Goals, *Computer*, May 1975.

Ross, D.T., and Schoman, K.E., Structured Analysis for Requirements Definition, *IEEE Transactions on Software Engineering*, January 1977.

Stevens, W.P., Myers, G.J., and Constantine, L.L., Structured Designs, *IBM Systems Journal*, Vol. 13, No. 2, 1974.

Thayer, R.H., Pyster, A., and Wood, R., The Challenge of Software Engineering Project Management, *Computer*, August 1980.

Wirth, N., On the Composition of Well-Structured Programs, *Computing Surveys*, Vol. 6, No. 4, December 1974.

Zelkowitz, M.V., Perspectives on Software Engineering, *Computing Surveys*, Vol. 10, No. 2, June 1978.

DIVERSEN

Caglayan, M.U., van Wigen, J.W., and Huskey, V.R., On the Poetic Connection of Ada, *Communications of the ACM*, Vol. 24, No. 8, August 1981.

Foss, D.J., and Hakes, D.T., *Psycholinguistics: An Introduction to the Psychology of Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

Gall, J., *Systemantics: How Systems Work and Especially How They Fail*, Times Books, New York, 1977.

Hoare, C.A.R., *Professionalism*. Paper presented to British Computing Society, London, July 1981.

Hofstadter, D.R., *Godel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1979.

Miller, G.A., The Magical Number Seven, Plus or Minus Two, *Psychological Review*, Vol. 63, No. 2, March 1956.

Moore, D.L., *Ada, Countess of Lovelace: Byron's Legitimate Daughter*, Harper & Row, New York, 1977.

Wegner, P., Ada - The Poetic Connection, *Communications of the ACM*, Vol. 24, No. 5, May 1981.

Weinberg, G.M., *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.

INDEX

ACADEMIC SERVICE INFORMATICA UITGAVEN

AUTOMATISERING EN COMPUTERS

Computers en onze samenleving - M.A. Arbib
Computers in de negentiger jaren - G.L. Simons
De informatiemaatschappij - Jan Everink
Basiskennis informatieverwerking - Jan Everink
AIV, Automatisering van de informatieverzorging - Th.J.G. Derksen en H.W. Crins
Organisatie, informatie en computers - D.M. Kroenke
De Viewdata revolutie - S. Fedida en R. Malik

MICROCOMPUTERS

Microcomputers thuis en op school - K.P. Goldberg en R.D. Sherwood
Bouw zelf een Expertsysteem in BASIC - C. Naylor
Programmeercursus Microsoft BASIC - Nok van Veen
Werken met bestanden in BASIC - L. Finkel en J.R. Brown
40 Grafische programma's voor de Commodore 64 - M. Sutter
Doe-het-zelf programma's op de Commodore 64 - D. Kreutner
Programmeercursus BASIC op de Commodore 64 - Nok van Veen
TRS-80 BASIC - Bob Albrecht e.a.
TRS-80 BASIC voor gevorderden - Don Inman e.a.
Exidy sorcerer en BASIC - Nok van Veen e.a.
40 Grafische programma's voor de Electron en BBC - M. Sutter
Het Electron en BBC Micro boek - Jim McGregor en Alan Watt
Ontdek de ZX-Spectrum - Tim Hartnell
Werken met bestanden op de Apple - L. Finkel en J.R. Brown
Programmeercursus Applesoft BASIC - Nok van Veen
40 Grafische programma's voor de Apple II,IIe,IIc - M. Sutter
40 Grafische programma's in MSX BASIC - M. Sutter
Programmeercursus MSX BASIC - Nok van Veen

MICROPROCESSORS EN ASSEMBLEERTALEN

Procescomputers, basisbegrippen - dr.ir. J.E. Rooda en ir. W.C. Boot
Cursus Z-80 assembleertaal - Roger Hutty
6502 Assembleertaal en machinecode voor beginners - A.P. Stephenson

BESTURINGSSYSTEMEN

Inleiding besturingssystemen - A.M. Lister
Systeemprogrammatuur en software-ontwikkeling voor microcomputers - E. Verhulst
Bedrijfssystemen - EIT-serie, deel 4
CP/M het operating system voor microcomputers - J.N. Fernandez en R. Ashley
CP/M 86 - Nok van Veen
CP/M voor gevorderden - A. Clarke e.a.
PC DOS, het besturingssysteem van de IBM PC - R. Ashley en J.N. Fernandez
MS/DOS, het besturingssysteem voor 16 bit microcomputers - R. Ashley en J.N. Fernandez
UNIX, het standaard operating system - G.J.M. Austen en H.J. Thomassen
Werken met UNIX - Brian W. Kernighan en Rob Pike

PERSONAL COMPUTERS

Het werken met bestanden op de IBM PC - L. Finkel en J.R. Brown
De IBM PC en zijn toepassingen - Laurence Press
40 Grafische programma's voor de IBM PC - M. Sutter
Werken met VisiCalc - C. Klitzner en M.J. Plociak
Multiplan, een hulpmiddel bij de bedrijfsvoering - D.F. Cobb e.a.
Multiplan diskettes
Werken met Lotus 1-2-3 - D. Cobb en G. LeBlond

PROGRAMMEREN

Een methode van programmeren - prof.dr. Edsger W. Dijkstra en ir. W.H.J. Feijen
Programmeren, het ontwerpen van algoritmen (met Pascal) - ir. J.J. van Amstel
Inleiding tot het programmeren, deel 1 - ir. J.J. van Amstel
Inleiding tot het programmeren, deel 2 - ir. J.J. van Amstel
Programmeren, deel 2: van analyse tot algoritme - prof.drs. C. Bron
Inleiding programmeren en programmeertechnieken - EIT-serie, deel 1
Het Groot Pascal Spreuken Boek - H.F. Ledgard e.a.
JSP - Jackson Struktureel Programmeren - Henk Jansen
JSP Uitwerkingenboek - Henk Jansen

PROGRAMMEERTALEN

Aspecten van programmeertalen - ir. J.J. van Amstel en ir. J.A.A.M. Poirters
Programmeertalen, een inleiding - ir. J.J. van Amstel e.a.
BASIC - EIT-serie, deel 3
Cursus BASIC, een practicum-handleiding voor BASIC op de PRIME - ir. R. Bloothoofd e.a.
Cursus Pascal - prof.dr. A. van der Sluis en drs. C.A.C. Görts
Cursus eenvoudig Pascal - prof.dr. A. van der Sluis en drs. C.A.C. Görts
Inleiding programmeren in Pascal - C. van de Wijgaart
Systeemontwikkeling met Ada - Grady Booch
Cursus COBOL - A. Parkin
Cursus FORTRAN 77 - J.N.P. Hume en R.C. Holt
Aanvulling cursus FORTRAN 77 voor PRIME-computers - ing. J.M. den Haan
De programmeertaal C - ir. L. Ammeraal
Flitsend Forth - Alan Winfield
Programmeren in LISP - prof.dr. L.L. Steels

GEGEVENSSTRUCTUREN EN BESTANDSORGANISATIE

Informatiestructuren, bestandsorganisatie en bestandsontwerp - EIT-serie, deel 5
Programmeren, het ontwerpen van datastructuren en algoritmen - ir. J.J. van Amstel e.a.
Bestandsorganisatie - prof.dr. R.J. Lunbeck en drs. F. Remmen

DATABASE EN GEGEVENSANALYSE

Database, een inleiding - C.J. Date
Databases - drs. F. Remmen
Gegevensanalyse - R.P. Langerhorst

INFORMATIE-ANALYSE EN SYSTEEMONTWERP

Effectieve toepassingen van computers - M. Peltu
Voorbereiding van computertoepassingen - prof.dr. A.B. Frielink
Systeemontwikkeling volgens SDM - H.B. Eilers
Samenvatting SDM - Pandata
Informatie-analyse volgens NIAM - J.J.V.R. Wintraecken
Evaluation of methods and techniques for the analysis, design and implementation of information systems - ed. J. Blank en M.J. Krijger
Inleiding systeemanalyse, systeemontwerp - W.S. Davis
Systeemontwikkeling Zonder Zorgen - Paul T. Ward
Het ontwerpen van interactieve toepassingen en computernetwerken - J.A. Scheltens
EDP Audit - prof.dr. C. de Backer
Prototyping, een instrument voor systeemontwerpers - ed. T. Hoenderkamp en H.G. Sol
Simulatie, een moderne methode van onderzoek - drs. S.K. Boersma en ir. T. Hoenderkamp

EXPERT SYSTEMEN EN KUNSTMATIGE INTELLIGENTIE

Computerschaak - H.J. van den Herik
Expert systemen - Henk de Swaan Arons en Peter van Lith

THEORETISCHE INFORMATICA EN SYSTEEMPROGRAMMATUUR

Informatica, een theoretische inleiding - dr. L.P.J. Groenewegen en prof.dr. A. Ollongren
Systeemprogrammatuur - drs. H. Alblas
Vertalerbouw - H. Alblas e.a.

AANVERWANTE ONDERWERPEN EN OVERIGE TITELS

Lineaire programmering als hulpmiddel bij de besluitvorming - prof.dr. S.W. Douma
Inleiding programmeren - prof.dr. R.J. Lunbeck
Analyse van informatiebehoeften en de inhoudsbeschrijving van een databank - prof.dr. P.G. Bosch en ir. H.M. Heemskerk
Gegevensstructuren - R. Engmann e.a.
Cases en Uitwerkingenboek bij Cases - prof.dr. P.G. Bosch en H.A. te Rijdt
De tekstmachine - dr. M. Boot en drs. H. Koppelaar
Abstracte automaten en grammatica's - prof.dr. A. Ollongren en ir. Th.P. van der Weide
Onderneming en overheid in systeem-dynamisch perspectief - red. A.F.G. Hanken e.a.
Simulatie en sociale systemen - red. J.L.A. Geurts en J.H.L. Oud
Struktuur en stijl in COBOL - ir. E. Dürr en dr.ir. F. Mulder
Cursus ALGOL 60 - prof.dr. A. van der Sluis en drs. C.A.C. Görts

Het verschijnsel dat er meestal veel meer tijd en geld gaat zitten in het ontwikkelen van geautomatiseerde systemen dan men tevoren gedacht en gehoopt had, wordt met een zeker gevoel voor drama omschreven als de 'software crisis'.
Een crisis ontstaat bij gebrek aan geschikt gereedschap voor het beheersen en besturen van een proces en in dit opzicht hebben veel automatiseringsprojecten inderdaad nog steeds iets crisis-achtigs.
In dit boek wordt de programmeertaal Ada behandeld in het licht van deze problematiek. Het gaat hier zeker niet alleen om de behandeling van 'alweer een nieuwe programmeertaal', maar het doel van de schrijver is juist geweest een complete gereedschapsset te bieden, met behulp waarvan ook zeer complexe en grote programma's op een beheerste en systematische wijze door een team van programmeurs ontwikkeld kunnen worden.

Programmeurs vinden in dit boek een volledige en heldere behandeling van de programmeertaal Ada.

Projectleiders vinden tal van aanwijzingen voor het beheersbaar houden van het ontwikkelingsproces.

Docenten aan universiteiten, hogescholen en instituten voor hoger beroepsonderwijs zullen de didactische opzet en de instructieve opgaven waarderen.

Iedereen met belangstelling voor de stormachtige ontwikkelingen in de informatica vindt in dit boek een actuele samenvatting van de huidige ideeën over programmeertalen en systeemontwikkeling.

De auteur Grady Booch is een Ada-expert en ontwikkelde een Ada-cursus in opdracht van het Ada Joint Program Office (AJPO), die inmiddels door meer dan 2500 cursisten gevolgd is. Booch publiceerde een twintigtal artikelen over Ada en programma-ontwikkelingsmethoden.

ISBN 90 6233 133 5